A Restful Architecture for Web-based Smart Homes using Request Queues

Technical Report No. TR-12-5

Andreas Kamilaris and Andreas Pitsillides
kami@cs.ucy.ac.cy, andreas.pitsillides@ucy.ac.cy

NETworks Research Laboratory
Department of Computer Science
University of Cyprus

June 2012

Abstract

This technical report describes the architecture of a Web-based smart home framework, which may be used as the backbone of future smart home deployments. This framework has been designed to support multiple family members and concurrent invocation of embedded home appliances, sensors and actuators. A fundamental element of the framework's architecture is the use of request queues, for better managing the communication with home devices, ensuring reliability, fault-tolerance and time efficiency. By means of the requests queues, prioritized requests may be easily supported, by transforming the queues into priority heaps. The framework has been designed following the REST architectural style, in order to offer a Web-based behaviour and address the increased heterogeneity of resource-constrained home devices. Through this report, the design and implementation of this Web-based framework for smart homes is presented, along with an analysis of its important components. Finally, a technical evaluation is provided to demonstrate the functionality of the system both in typical and heavy workloads.

Table of contents

List of Figures

Chapter 1

Introduction

Technological advancements such as sensor networks, short-range wireless communications and real-time localization are becoming largely common, allowing the Internet to penetrate in embedded computing. An *Internet of Things* (IoT) [4, 12] is becoming possible, where everyday objects are uniquely addressable and interconnected.

Recent efforts for porting the IP stack on embedded devices [2, 8] and the introduction of IPv6, which provides extremely large addressing capabilities, are expected to facilitate the merging of the physical and the digital world, through the Internet.

Building upon the notion of the IoT, the *Web of Things* (WoT) [16, 7] reuses well-accepted and understood Web principles to interconnect the expanding ecosystem of embedded devices. While the IoT focuses on interconnecting devices at the network layer, the WoT can be seen as a promising practice to achieve interoperability at the application layer. It is about taking the Web as we know it and extending it so that anyone can plug devices into it.

Household appliances are also being affected by embedded technology. They are being equipped with embedded micro-controllers and wireless transceivers, offering communication capabilities and providing smart behavior. These augmented appliances, when interconnected, they can form wireless networks, extending residential areas into smart home environments. We believe that future generations of smart homes will be massively equipped with networked household appliances, smart meters, smart power outlets, sensors and actuators. These devices will be much more ubiquitous, integrated in our everyday lives and can even be highly mobile. Their management would require a high degree of flexibility.

Unfortunately, the existing solutions for smart homes are currently highly heterogeneous and vendor-specific. A common ground does not exist to support interoperability between devices and services from different manufacturers. The Web, as an ubiquitous protocol that scales particularly well, may be appropriate to constitute the application layer in future smart home scenarios.

Related studies show that an IP-based approach for home automation can offer equivalent performance with related traditional home automation standards, while concepts from the Web bring convenience to developers and users [11, 5]. In this report, the design, architecture and development of a Web-based application framework for smart homes is presented, aiming to address the issue of heterogeneity in embedded technology, offering satisfactory performance.

Chapter 2

A Web-based Application Framework for Smart Homes

This chapter describes the general architecture, design and development of a lightweight, Web-oriented application framework [10, 9] that provides uniform access to heterogeneous embedded devices via standard HTTP calls.

2.1 The Architecture of the Application Framework

Figure 2.1 depicts the general architecture of this framework. It follows a layered model and it is composed of three principal layers: *Device Layer*, which is responsible for the management and control of embedded devices, *Control Layer*, which is the central processing unit of the system and *Presentation Layer*, which generates dynamically a representation of the available devices and their corresponding services to the Web, enabling the uniform interaction with them over a Restful interface.

Each time a new device is discovered, a new thread dedicated to the physical device is created. A *Resource Registry* maintains the services offered by this device as well as information how to properly invoke them. A *Request Queue* is also attached to each thread, to enqueue concurrent requests to it. Requests are stored in a FIFO manner and are transmitted sequentially to the device. Whenever the queue "suspect" that the request might have been lost, it retransmits it immediately. In this way, transmission failures are masked effectively. The mechanism for sensing that a transmission could have failed is explained in Chapter 4.

Devices module holds a list of all available devices and their services, inside a *Device Registry*, facilitating the Web representation of the smart home status. *Driver* holds the technology-specific drivers for enabling communication with embedded devices, by sending/receiving requests to/from them. *Control Layer* hosts the Core module, which runs in the background and maintains the system's threads.

Finally, *Presentation Layer* represents the access point to the framework from the Web. A *Web Server* allows real-time interaction between Web clients and their home environment. A *REST Engine*, implemented by means of Restlet[1], ensures a Restful system behaviour. Thanks to REST, Web clients can easily explore available devices and their corresponding services by clicking links, as they would browse Web pages.

[1] http://www.restlet.org/

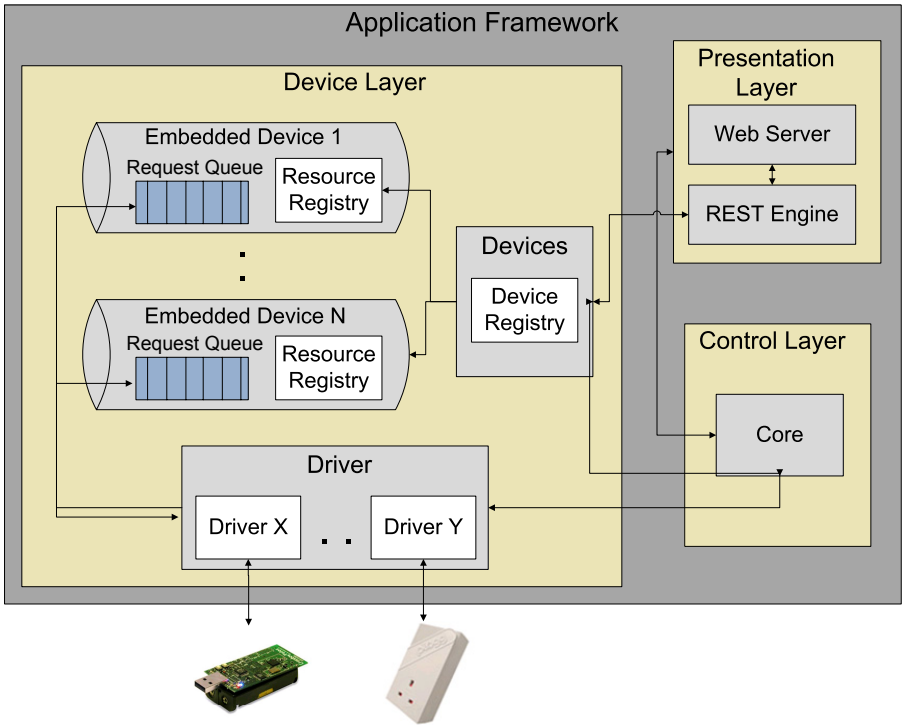

Figure 2.1: The general architecture of the application framework.

2.2 Reasoning about REST

REST was adopted as the foundational architectural style for communicating both with Web clients and physical devices. REST may be suitable for constrained environments due to its simplicity and flexibility. It can guarantee interoperability and a smooth transition from the Web to the home environment.

Undoubtedly, many standards offer interoperability between heterogeneous machines. Their main difference from REST is that they include a rather complicated set of protocols and guidelines for interaction between devices, while REST suggests to use merely the HTTP protocol and basic Web principles to address heterogeneity.

A recent study [6] compares REST with its main opponent, big Web services (WS-*) [1]. WS-* are a set of complex standards and specifications for enterprise application integration. The study suggests that REST is easier to learn and understand, more lightweight, flexible and scalable, better for heterogeneous environments while WS-* has better security features, but is more complex to be used, being more appropriate for business applications.

Extensible Messaging and Presence Protocol (XMPP)[2] is an open XML-based communications technology that could be applied, however it is also heavy for use with embedded devices. JXTA is another alternative, but it is more of an overlay network, not directly related to the concept of the Web.

Constrained Application Protocol (CoAP) [14] is an IETF effort to develop a Web transfer protocol for use with resource-limited physical devices, aiming to realize the REST architecture in constrained networks. Even though it claims to be a simple protocol with low overhead, it is yet another attempt for standardization, which does not comply to the general openness of the Web. Nonetheless, it has some promising features such as asynchronous message exchange

[2] http://xmpp.org/

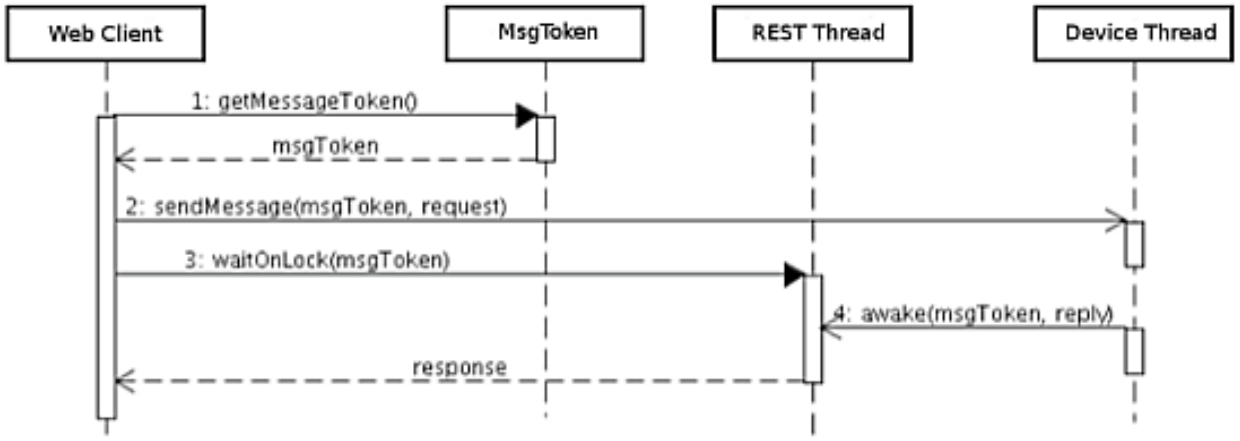

Figure 2.2: Transition between (synchronous) Web functionality and (asynchronous) smart home operation and vice-versa.

and low header overhead. It could be considered in the future when it possibly becomes more mature.

2.3 Synchronous/Asynchronous Translation

One of the major criticisms against the use of REST in the area of smart homes spans from the fact that HTTP was designed for synchronous requests. This model does not generally suit the requirements of most sensor network applications.

However, we achieved to overcome this issue by mapping a normal HTTP request, coming from a Web client at the Web server module of the application framework, to an asynchronous operation which happens on the device layer of the framework.

Synchronizers were employed, in order to transform asynchronous behavior into a synchronous one. For each Web request, the framework creates a new REST thread, associated with a unique message token (*requestID*). This requestID is encapsulated into the original Web request and the request is forwarded to the thread of the relevant home device. There, it obtains a lock on a unique synchronizer token, labeled using the requestID.

As soon as the response from the embedded device arrives, the synchronizer lock searches for the request with requestID and the appropriate waiting thread that represents the Web request is awakened. In this way, the HTTP response is then sent back to the initial client. The duration of time needed to complete the Web request is highly dependent of the capabilities of the home device that is invoked.

Figure 2.2 shows the sequence diagram for this synchronous/asynchronous translation.

1. The client's Web request obtains a unique message token (requestID).
2. Together with the token, the request is sent asynchronously to the thread of the appropriate home device.
3. The request obtains (using the message token) a lock on its REST thread and puts itself to sleep.

4. The message arrives from the physical device. The framework awakes the REST thread of the Web request that has the correct message token.

5. The Web response message is forwarded back to the initial client.

2.4 Integrating a Web Cache

In general, a Web cache is an intermediary between Web servers and clients, monitoring incoming requests and saving copies of the responses for itself. In case there are subsequent requests for the same URL, the cached response is harnessed, instead of asking the origin server for it again.

A lightweight Web cache is included on the application framework. In this case, Web servers are the home devices, Web clients are the residents of the smart home and a gateway cache is installed on the framework. The caching mechanism may be used only for GET requests since they are requests that do not alter the state of some resource, according to the REST specifications. The cache uses the expiration model, which is a way for the server to say how long a requested resource stays fresh.

Whenever a client requests a GET service, offered by some specific device, which was recently invoked by another client, the result is automatically retrieved from the cache, instead of querying the physical device again. This is called a cache hit, since the measurement was successfully retrieved from the cache. In case the cache did not contain a fresh answer, this would be a cache miss.

The amount of time some measurement is considered valid (fresh), depends on some user-defined parameter. At the technical evaluation in Chapter 7, this parameter is set to 10 seconds. Obviously, a larger amount of time implies more cache hits, however, the results may not be as fresh. On the contrary, smaller cache validity times mean more fresh results but also increased cache misses.

Chapter 3

Experimental Setup

In the next sections, we explain the experimental setup that will be used in the next chapters, to analyze the application framework's operation and to perform a technical evaluation. Additionally, the embedded devices that are included in the evaluation efforts are described.

3.1 Emulating a Smart Home Scenario

In order to analyze the request queues, realistic traffic from Web clients who represent family members needs to be created. Home residents are modeled as software agents who reside on the same computer as the application framework. Therefore, network delay is negligible. Using Restlet, a unique TCP socket is assigned to each agent.

The arrival rate of family members at the framework is modeled by the exponential distribution. The exponential distribution is preferred to simulate real-world applications that deal with the occurrence of events, such as the arrival of customers at a bank. The parameter λ denotes the arrival rate of the clients. For example, when $\lambda = 2$, this means that two family members arrive at the framework every second. The operations of each agent include the *random* selection of a sensor mote, as well as the random invocation of a Restful Web service offered by the device.

3.2 Embedded Devices inside the Smart Home

Telosb sensor motes were employed during the experiments in the smart home, for sensing the environmental conditions inside the home environment.

These motes [13] are equipped with a 250kbps, 2.4GHz, IEEE802.15.4[1]-compliant Chipcon CC2420 Radio, integrated onboard antenna and a 8MHz TI MSP430 microcontroller with 10 kB RAM. They integrate by default temperature, humidity and illumination sensors, but can also support other sensors (e.g. noise, pollution). A typical Telosb mote is illustrated in Figure 3.1 (left). In the right side of the figure, the internal design of this sensor platform is presented.

We programmed the motes using TinyOS 2.x[2], which is an operating system designed for low-power wireless devices. By means of Blip[3], which is an implementation of the 6LoWPAN stack for TinyOS, we exposed the default sensing capabilities of the motes as Restful Web services, transforming them into embedded Web servers. This means that these Web-enabled

[1] http://www.ieee802.org/15/pub/TG4.html
[2] http://www.tinyos.net/
[3] http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip

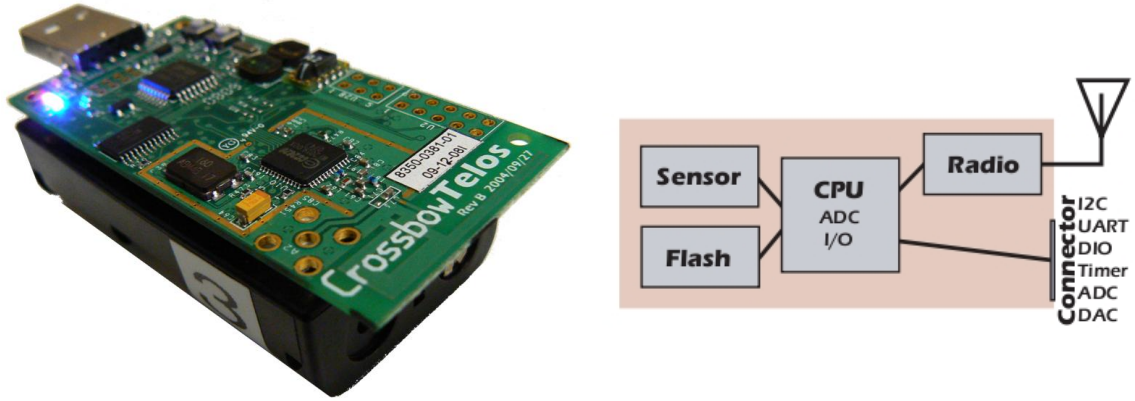

Figure 3.1: A Telosb sensor mote and its internal design.

embedded devices expose their services under a Restful interface, while communication is based on standard HTTP calls.

6LoWPAN [3] is an adaption layer that allows efficient IPv6 communication over the IEEE 802.15.4 standard. Through 6LoWPAN, an IPv6-enabled multi-hop wireless sensor network (WSN) was formed, consisting of Telosb motes. Thus, by deploying these devices around the smart home, the environmental conditions inside the house could be sensed in real-time, while the multi-hop topology would enable the wireless propagation of the measurements from the remote sensors to the base station.

Chapter 4

Analyzing the Functionality of Request Queues

The application framework associates each home device with a request queue, for supporting simultaneous family members to send concurrently HTTP requests to them. The request queues have the responsibility of forwarding requests to the devices sequentially, according to the request time of arrival. As soon as a response arrives, the corresponding request is removed and the next request is forwarded to the head of the queue.

Newly incoming requests need to wait in the queue for their turn, in order to be executed. Figure 4.1 shows a snapshot of the request queue load, during a typical framework's operation, for three different scenarios of arrival rates λ, of requests per second. As the arrival rate of requests increases, the size of the queue is increased as well. This size may be up to 4, when λ equals 1.5 and up to 7, when $\lambda = 2$. The waiting time at the queue is a significant source of delay, as depicted in Figure 4.2 for the case when $\lambda = 1.5$. Requests need to wait few seconds when the size of the queue is more than 2.

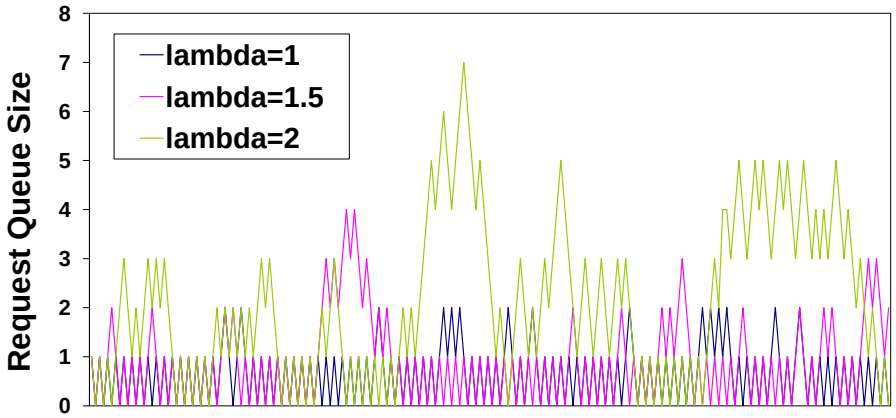

Figure 4.1: Load at the request queue of some sensor device during its operation.

During the framework's operation, in case some transmission message is lost, the queue waits some amount of time and then it retransmits the message. This amount of time is crucial for the system's performance, since fast retransmissions would increase the service rate of the requests. However, retransmissions that happen too early may cause collisions and make the system inefficient. Thus, fine-tuning the request queue retransmission interval is important for the framework's operation. In the following subsections, the main focus is to analyze the queue

behaviour, and properly set the value of the retransmission interval. From now on, we would refer to this parameter as α.

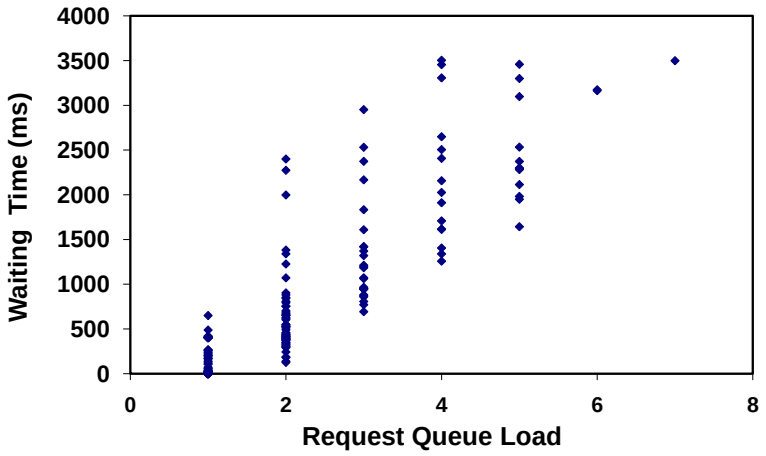

Figure 4.2: Waiting times of the requests at the request queue of some sensor device.

4.1 Searching for an Effective Request Queue Interval

The experimental setup for analyzing the queue behaviour involves two sensor devices in a star topology, 5 meters distance from the base station and from each other. A small number of devices was selected, as the aim was to observe the queues when having considerable load.

Before proceeding to the first experiment, we need to identify the different time-related metrics used in the analysis procedure:

- Average Request-Response Time or Response Time. The average amount of time needed from the creation of a request by a Web client to the arrival of the response, transmitted by a sensor device.
- Round Trip Time (RTT). The average amount of time from the wireless transmission of a request until the arrival of the response.
- Waiting Time. The amount of time a request waits at the queue for its turn.

Obviously, $Response\ Time = Waiting\ Time + Round\ Trip\ Time$.

The first experiment studies the effect of transmission failures on the request queues, using $\lambda = 1$. By default, the transmission failures in our smart home deployment are around 5%, so we emulated additional failures by manually dropping packets at the driver module. Figure 4.3 presents the response times in relation to the queue interval α, considering different percentages of failed transmissions.

As the transmission failures increase, response times are also increased. In all scenarios of different percentages of transmission failures, $\alpha = 600\ ms$ yields the best response times. As the parameter α increases, response times are increased as well. An important thing to notice is that as the percentage of transmission failures increases, the slopes of the graphs become larger. This fact denotes that the request queue mechanism becomes more significant when transmission failures exist in some pervasive scenario.

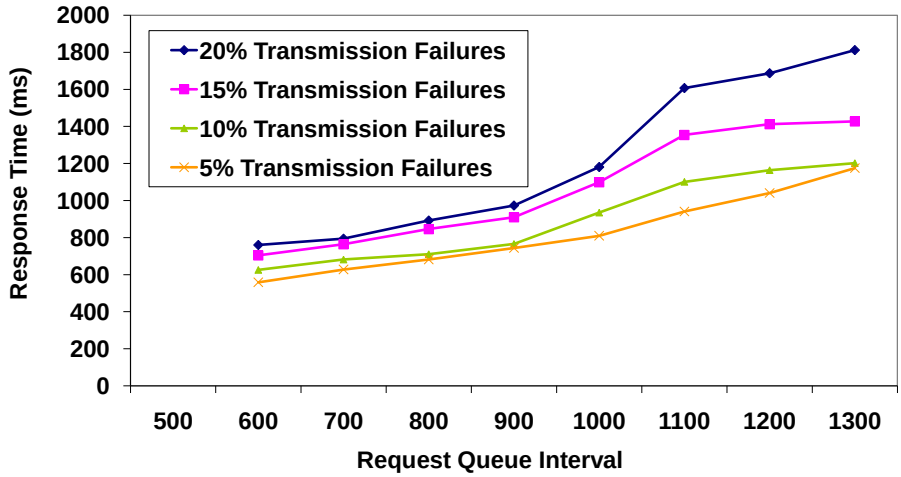

Figure 4.3: The effect of transmission failures on the request queues.

When α is less than 600 ms, the system does not operate correctly. As displayed in Figure 4.4, the percentage of retransmission attempts -in relation to the total number of client requests- grows exponentially as α decreases, since the queue incorrectly believes that the request messages keep failing. Continuous retransmissions cause the sensor devices to malfunction and average response times become very large as well.

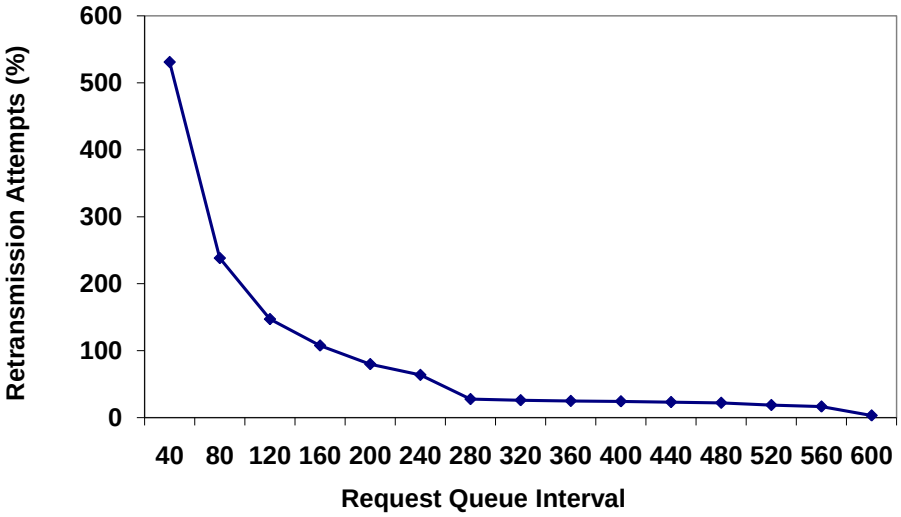

Figure 4.4: Retransmission attempts in low values of the request queue retransmission interval.

The next experiment examines the interval α in varied workload. In this and the next experiments, the default case of 5% transmission failures is used. This is realistic for typical home environments. The results of this experiment are displayed in Figure 4.5.

Low traffic implies a small percentage of retransmissions. Thus, the influence of the request queue in the overall system performance is small. However, as the traffic increases, the retrans-

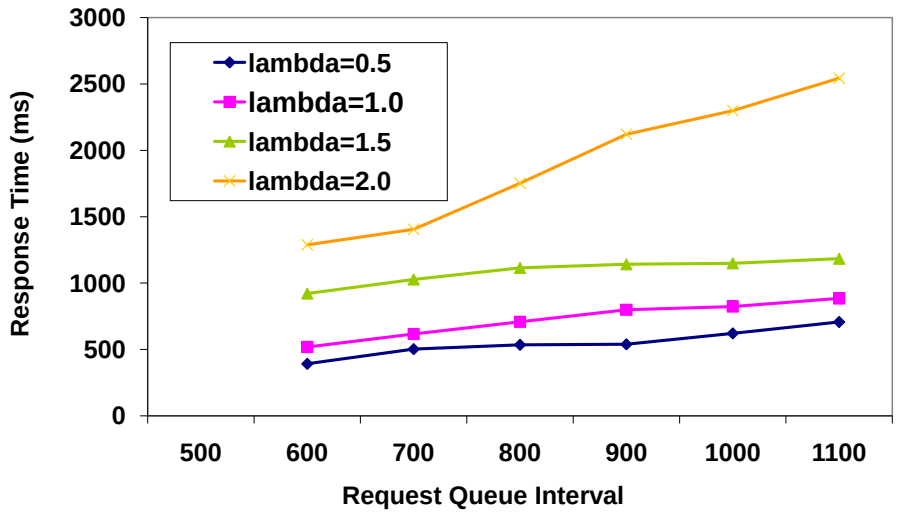

Figure 4.5: The effect of varied workload on the request queues.

mission interval α becomes more important, especially when $\lambda = 2$. Also in this experiment, the most appropriate value for α is 600 ms.

4.2 An Equation for Setting the Request Queue Interval

Considering that the average RTT time in the one-hop communication with the sensor devices is 530 ms, with a standard deviation equal to 55, we assume that the appropriate value of α equals the RTT time plus the standard deviation value. To check whether this assumption is correct, we performed the last experiment, which investigates the influence of RTT time on the request queue, as shown in Figure 4.6 for $\lambda = 1$. To increase RTT time, we programmed the TinyOS application which was installed on the sensor devices to wait an extra delay before transmitting a response back to the framework.

As the RTT delay increases, the best value for α equals the RTT time plus the standard deviation value, in all cases of different delays. Therefore, we recommend that the α value must be set according to the following equation:

$$\alpha = RTT + St.\ Deviation \tag{4.1}$$

Of course, RTT times and standard deviation values are learned from the device thread after observing the physical device for some amount of time. Therefore, the best approach is to set initially α to a larger value, leaving a "safe margin" to avoid producing failed attempts. During the device operation, the value of α may be fine-tuned accordingly, through observation of the time intervals encountered.

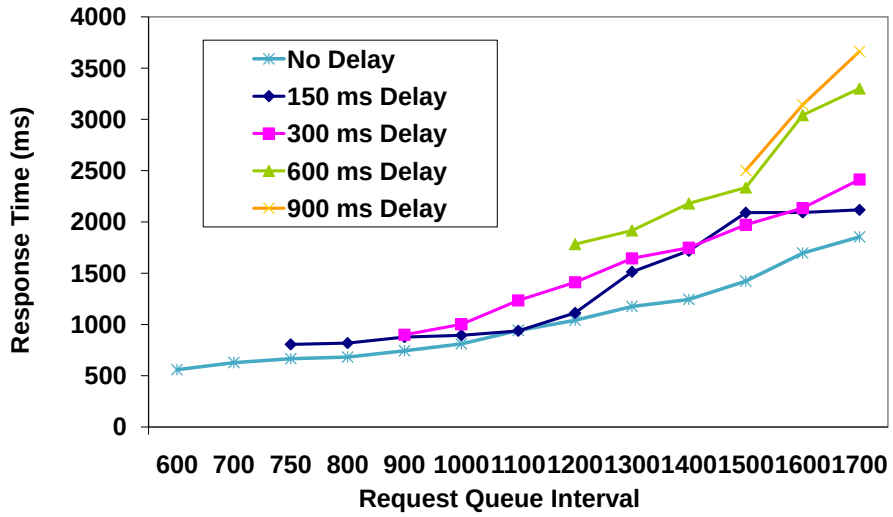

Figure 4.6: The effect of round trip time on the request queue.

Chapter 5

Using Request Queues for Enhancing the Performance of Smart Home Operations

By using request queues at the application framework for managing the communication with embedded devices, numerous benefits can be obtained. We list some of these benefits in the following sections.

5.1 Multi-Client Support and Masking Transmission Failures

By employing request queues, multiple family members may interact simultaneously with their home environment. Even though their requests could be delayed in increased traffic conditions, it is guaranteed that they would eventually be satisfied.

Additionally, as mentioned before, transmission failures are masked. As soon as a failure is "suspected" by the queue, the request is immediately retransmitted to the device.

5.2 Estimating Current Response Times

Another important aspect is the capability of predicting the average request-response time. To predict the (current) amount of time needed for satisfying some request, the following equation can be used:

$$Response\ Time = Request\ Queue\ Size * RTT * (1 + p_f) \tag{5.1}$$

where p_f is the probability of transmission failures towards the specific home device.

5.3 Avoiding Overloading of the Request Queues

the stability condition of each request queue may be roughly estimated by observing the arrival rate of incoming requests λ. To achieve stability, it must be ensured that service time is less than the inter-arrival time, i.e. $RTT < \frac{1}{\lambda}$. In our earlier experiments with $RTT = 530\ ms$, this implies that $\lambda < 1.88$ *for each* sensor device.

In theory, this may be generalized for x devices, assuming that requests are distributed uniformly to them. In this case, the total arrival rate λ may reach $x * 1.88$ requests per second. However, in practice, large arrival rates can not be supported because of other factors

affecting the system's behaviour such as the framework's processing delay and the transmission throughput of the sensor mote acting as the proxy between the Web and the physical environment.

In a real-world setting, the stability condition might be more appropriate to be averaged over 30 minutes or some hours, to avoid dropping requests in unexpected rush hours that last only some minutes.

5.4 Serving Increased Traffic through Redundancy of Embedded Devices

Another possibility would be to use redundant sensor devices to effectively serve increasing traffic. An equivalent example from real life concerns a queue at a bank, which employs a number of employees, in order to serve efficiently its customers.

In this case, a module acting as a load balancer may be considered for managing a network of queues. This module could be placed on the framework between the driver module and the device threads.

Chapter 6

Supporting Priorities

The use of request queues may easily support prioritized requests. Priorities are considered in cases when an order of importance or urgency needs to be maintained. Typical categories of priorities may be *low*, *normal* and *high*. High-priority requests are the most important, while low-priority requests the least significant.

In smart home applications, using prioritized requests may be beneficial when some urgent operation needs to be performed as fast as possible. This operation may involve triggering the alarm when a thief breaks into the house or switching off an electric appliance that consumes excessive amounts of energy.

The request queues may be easily transformed into priority queues by converting the queues into priority heaps. A key characteristic of binary heaps is that they keep at the head of their binary tree the request that has the highest priority. The algorithm we used for defining priorities in the smart home ecosystem is described in Algorithm 1.

Algorithm 1 An algorithm for setting prioritized requests in the smart home.

1: Assign a priority to each request (low, normal or high).
2: Each prioritized request is translated into an integer value by the application framework, according to the following formula: low=1, normal=5 and high=10.
3: The priority queue selects for execution the request with the highest priority number.
4: To avoid starvation of low-priority requests, the priorities of all waiting requests are increased by 1 at every round, i.e. at a successful execution of some request.

To test the priority queues of the application framework, the emulation described in Section 3.1 was employed. Two sensor devices in a star topology were considered, with 5 meters distance from the base station and from each other. Figure 6.1 shows the response times of the prioritized requests according to the current size of the queue when each request appeared. This emulation was performed by setting $\lambda = 2$, including 10% high-priority requests, 10% low-priority and the rest with normal priority.

According to the figure, high-priority requests are executed first, with waiting times less than two seconds, even when the queue has a size equal to 9. On the contrary, low-priority requests need significantly more time to be executed, reaching 10 seconds in some cases. Table 6.1 shows the load conditions of the priority queue at each priority category. As expected, high-priority requests remain the least time in the queue, between 1-3 rounds. Low-priority requests stay in average 5 rounds in the queue, while requests with normal priority fluctuate between low- and high-priority requests.

The average waiting times of the priority categories when changing the percentage of high-

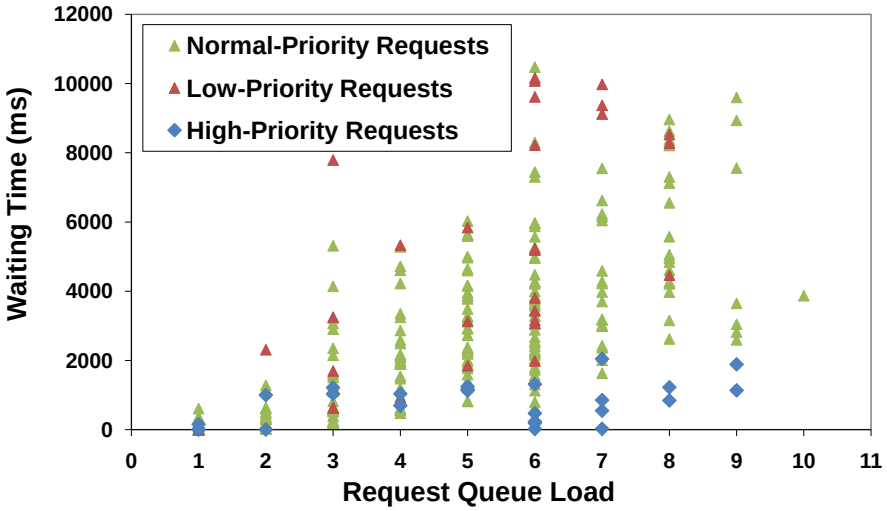

Figure 6.1: Waiting times for requests with different priorities and request queue conditions.

Priority	Min. Queue Load	Max. Queue Load	Avg. Load	St. Dev.
Low	1	12	5.40	3.808
Normal	1	9	3.24	2.421
High	1	3	1.44	1.003

Table 6.1: Priority queue load conditions at the different priority categories.

priority requests are presented in Figure 6.2. As the number of high-priority requests increases, their waiting time is likewise slightly increased. This happens because few high-priority requests might enter the queue concurrently. In such cases, a time priority is followed and the high-priority requests appearing later need to wait those appearing earlier. As the percentage of high-priority requests increases, also the waiting times of the low- and normal-priority requests are increased. This occurs since the high-priority requests are executed first, delaying the execution of lower-priority requests.

Figure 6.2 indicates that the waiting time of high-priority requests scales well, as their percentage increases. This happens by forcing lower-priority requests to stay longer in the queue, with an exponential growth of their waiting time, as high-priority requests are increased. Algorithm 1 may be easily extended to support more layers of priorities.

Offering support for priorities is not a native feature of the Web, even though it may be supported in the future. However, priorities can still be employed internally by the application framework, e.g. to favour the requests coming from specific family members or those marked by the users as more important in some rule-based, smart home application.

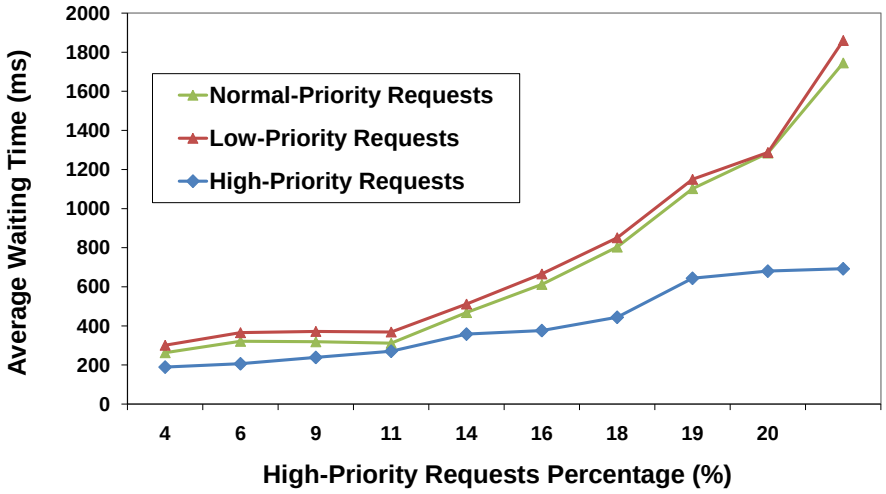

Figure 6.2: Average waiting times for different percentages of high-priority requests.

Chapter 7

Technical Evaluation

To test the smart home ecosystem, a common scenario was considered, in which a large family interacts concurrently with the home environment. This scenario demonstrates a request-response case with high traffic. Multiple virtual residents are assumed, interacting with the IPv6-enabled sensor devices of their smart home through the Web. The emulation described in Section 3.1 is followed.

Figure 7.1 depicts the system infrastructure of the smart home, which would be used in the evaluation procedure, deployed in a one-bedroom home.

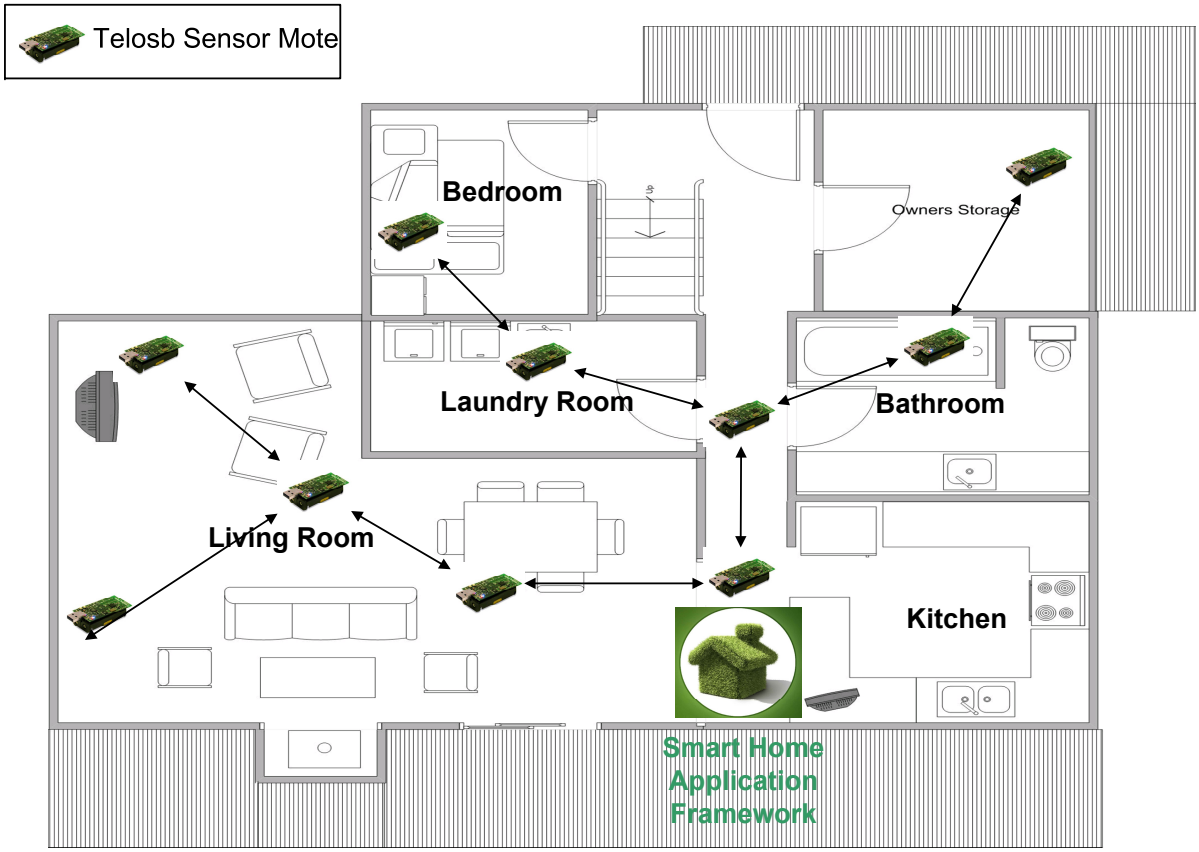

Figure 7.1: Experimental setup during technical evaluation.

In these experiments, the focus is on two performance metrics: average request-response

times (see Section 4.1); and energy consumption, i.e. the power consumed by the sensor devices during their operation.

An important configuration parameter that influences the measurement results is the transmission power used by the sensor motes. We set the transmission power at -25 dBm, to allow a 5-meters transmission range and facilitate a multi-hop operation of the WSN inside the home environment.

Another important aspect concerns the sizes of the messages involved in the interactions between the home devices and the application framework. The size of the 6LoWPAN messages was 128 bytes, while the messages at simulated REST, which was a light, simulated version of REST without the IPv6 overhead, had size 80 bytes.

7.1 A Single-Hop Topology

Considering a single-hop topology of sensor devices, 4 IPv6-enabled sensor motes were deployed in a star topology with 5 meters distance from the base station. A small number of sensors was selected, in order to test the system in a demanding case, measuring average request-response times in different workloads.

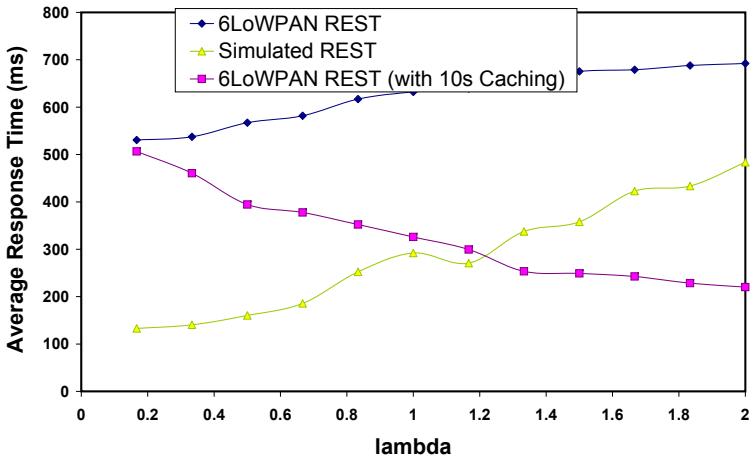

Figure 7.2: 6LoWPAN WSN single-hop completion time.

The results, shown in Figure 7.2, show that less than 700 ms were enough for satisfying the requests, even in the most extreme cases. Simulated REST proves to be more efficient than 6LoWPAN REST. However, when Web caching is employed (see Section 2.4), using only 10 seconds freshness time, 6LoWPAN REST executes faster than simulated REST in workloads that exceed an arrival rate $\lambda = 1.16$. In this case, the efficiency of 6LoWPAN REST depends on the percentage of cache hits at each test, as shown in Figure 7.3.

7.2 A Multi-Hop Topology

In most cases, single-hop topologies do not cover typical smart homes as the indoor range of a sensor mote is 20-30 meters. Hence, in this paper the execution times of the Web-based WSN were tested also in a multi-hop scenario. The four sensor motes were placed as leaf nodes in a three-hop distance from the base station, in a 2-3-4 topology, as shown in Figure 7.1. This

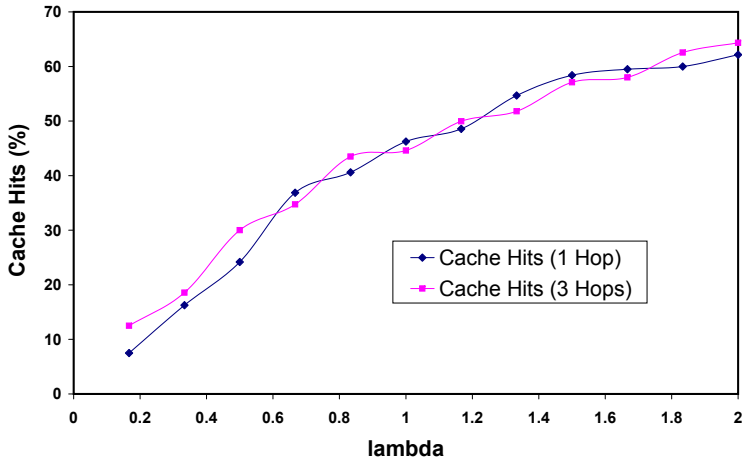

Figure 7.3: Cache success based on number of requests per minute.

topology was selected to stress the operation of the WSN under heavy workload. All sensor motes had 5 meters distance from their parent node. Various tests were performed with λ ranging from 0.2 to 2. Each test ran for 5 minutes.

The results can be observed in Figure 7.4. The average completion times are less than a second, even in the worst-case scenario. The multi-hop environment causes additional delays of around 170-200 ms for 6LoWPAN REST and 110-140 ms for simulated REST, compared to the single-hop tests. Similar to the single-hop scenario, 6LoWPAN REST with 10 seconds caching executes faster than simulated REST, when $\lambda > 0.8$.

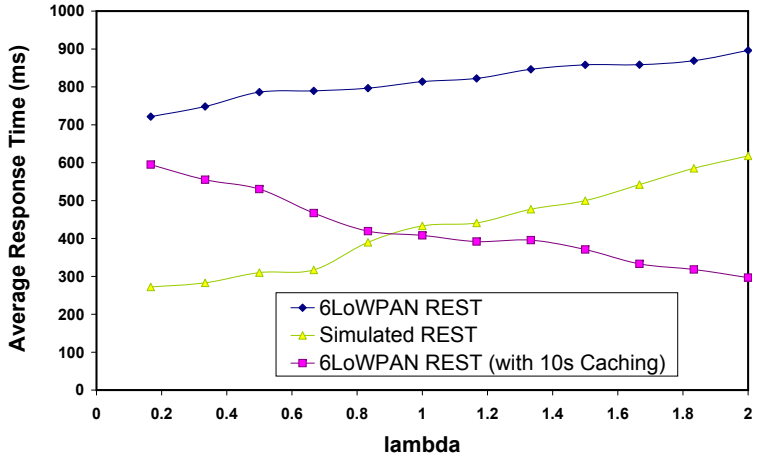

Figure 7.4: 6LoWPAN WSN multi-hop completion time.

Figure 7.3 shows the percentage of cache hits, in correlation to the arrival rate λ. As the cache hits increase, response time is effectively decreased. This fact indicates high scalability of the system. Numerous simultaneous Web requests can be handled only in a few hundreds of milliseconds. Comparing with the single-hop case, we conclude that cache hits do not depend on the sensor topology, being rather based on the workload and on the behavior of home residents at each test.

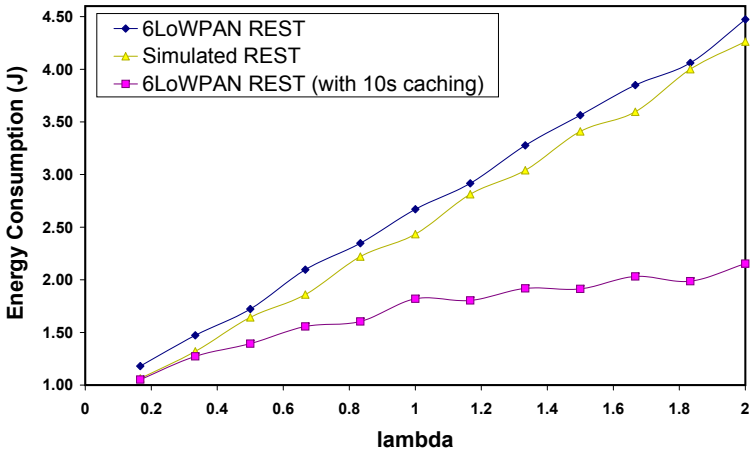

Figure 7.5: 6LoWPAN WSN energy performance.

7.3 Energy Performance of Sensor Devices

In this test, the energy performance of 6LoWPAN REST in comparison to simulated REST was studied, by employing the Avrora simulator [15]. Since Avrora simulates programs written for AVR microcontrollers, the TinyOS sensor application was tailored to run on Micaz motes. The power profiles of both mote types are listed in [13]. We focused on the leaf nodes of the three-hop scenario. Those are the sensors that satisfied the Web requests from home residents. The same tests were ran, assuming 5 meters node distance and 5 minutes execution time.

The energy consumption of one of the leaf sensor motes is displayed in Figure 7.5. The graph is for a single sensor but similar results have been obtained for the other leaf sensors. The simulation results for intermediate sensor motes in the three-hop topology follow a similar pattern.

Observing the figure, the difference in energy consumption between 6LoWPAN REST and simulated REST is small, around 2-12%. This variance happens due to the probabilistic selection of sensor motes by the residents. 6LoWPAN REST consumes slightly more energy due to the small overhead of packet size at 6LoWPAN. This small difference in energy consumption validates the statement that an IP-based stack for WSN does not necessarily consume more energy [8]. Both in 6LoWPAN REST and simulated REST, the consumption increases almost linearly.

When 6LoWPAN REST employs 10 seconds caching, it becomes much more energy-efficient in comparison to either 6LoWPAN REST or simulated REST, in workloads with $\lambda > 0.4$. This efficiency reaches 50% under heavy workloads. Under light workload, efficiency is still considerable, fluctuating between 15-25%. 6LoWPAN REST with caching scales particularly well as the number of requests increases.

We need to note that the results depend greatly on the transmission power setting. At some power settings, receiving (or idle listening) may consume more than transmitting. The link layer of the sensor nodes does not use any duty cycling, but it is always listening by default. The fact that sleep modes were not considered explains the rather increased values of energy consumption in the tests.

Chapter 8

Conclusion

In this paper, a Restful architecture for a Web-based application framework was described, which may constitute a foundational pillar towards a comprehensive energy-aware smart home. The novel design of the framework, employing request queues for handling the interaction of tenants with their home devices, enhances the overall performance of the system, offering reliability and fault-tolerance. The support of prioritized requests may be also important in future smart home deployments, when urgent requests must be executed as soon as possible.

Overall, a flexible application-level solution for home automation is proposed, based on combining existing Web technologies and reliable, well-studied Web techniques. The REST architectural style is followed in order to address heterogeneity of home devices.

Our technical evaluation indicates that the Web of Things can be successfully applied to smart homes. The evaluation efforts show that IPv6 can be well applied on sensor motes, while 6LoWPAN, especially when combined with Web caching, can even outperform (identical) applications that utilize the native messaging stack of embedded operating systems such as TinyOS. Thus, enhanced Web techniques such as Web caching can contribute in increasing the performance and scalability of smart home applications, reducing the power consumption of sensor devices, preserving their battery lifetime.

Concluding, we believe that the future in home automation is towards the Internet. Web technologies have the potential to become the future standards in smart home environments, towards an interoperable and sustainable world.

Bibliography

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures*. Springer, 2004.

[2] Adam Dunkels, Thiemo Voigt, and Juan Alonso. Making TCP/IP Viable for Wireless Sensor Networks. In *Proc. EWSN 2004*, Berlin, Germany, 2004.

[3] N. Kushalnagar et al. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals, August 2007. RFC 4919.

[4] Neil Gershenfeld, Raffi Krikorian, and Danny Cohen. The Internet of Things. *Scientific American*, 291(4):76–81, October 2004.

[5] Carles Gomez and Josep Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*, 48(6):92–101, 2010.

[6] Dominique Guinard, Iulia Ion, and Simon Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers Perspective. In *Proceedings of the 8th International ICST Conference on Mobile and Ubiquitous Systems (Mobiquitous 2011)*, Copenhagen, Denmark, December 2011.

[7] Dominique Guinard, Vlad Trifa, and Erik Wilde. Architecting a Mashable Open World Wide Web of Things. Technical Report No. 663, Dept. of Computer Science, ETH Zurich, February 2010.

[8] Jonathan W. Hui and David E. Culler. IP is dead, long live IP for wireless sensor networks. In *Proc. SenSys 2008*, pages 15–28, Raleigh, NC, USA, 2008. ACM.

[9] Andreas Kamilaris, Vlad Trifa, and Andreas Pitsillides. HomeWeb: An Application Framework for Web-based Smart Homes. In *18th International Conference on Telecommunications (ICT 2011)*, Ayia Napa, Cyprus, May 2011.

[10] Andreas Kamilaris, Vlad Trifa, and Andreas Pitsillides. The Smart Home meets the Web of Things. *International Journal of Ad Hoc and Ubiquitous Computing (IJAHUC), Special issue on The Smart Digital Home*, 7(3):145–154, 2011.

[11] Matthias Kovatsch, Markus Weiss, and Dominique Guinard. Embedding Internet Technology for Home Automation. In *Proceedings of ETFA 2010 (IEEE Conference on Emerging Technologies and Factory Automation)*, Bilbao, Spain, September 2010.

[12] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. Accepted for publication (Springer LNCS), August 2010.

[13] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN)*, Piscataway, NJ, USA, 2005. IEEE Press.

[14] Zach Shelby, Klaus Hartke, Carsten Bormann, and Brian Frank. Constrained Application Protocol (CoAP), March 2012. IETF Draft, draft-ietf-core-coap-09.

[15] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN)*, 2005.

[16] Erik Wilde. Putting things to REST. Technical Report UCB iSchool Report 2007-015, School of Information, UC Berkeley, November 2007.