Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RESEARCH GROUP FOR
Distributed Systems

Application Kernels for Smart Home Environments

Bachelor's Thesis

Gianluca Andrea Vinzens
08-910-606

vinzensg@student.ethz.ch

Distributed Systems Group
Bachelor's Thesis
ETH Zürich

Supervisors:
Dipl.-Ing. Matthias Kovatsch,
Prof. Dr. Friedemann Mattern

August 28, 2012

Abstract

Distributed heterogenous devices and sensors make networked embedded systems hard to program. Applications for the Internet of Things should be realized in a Web-like infrastructure, where devices and sensors export RESTful interfaces. We identify application classes in literature about distributed applications, amongst others on Wireless Sensor Networks (WSN), and provide reusable application modules. The collection of modules can be combined to different applications using Web-like mashups based on the Constrained Application Protocol (CoAP). All application classes identified in the literature can be realized using the created modules. A detailed case study shows that the modules fulfill all requirements of an actual application.

Keywords: Modules, Application Classes, Application Kernels, CoAP, Californium, Actinium

Affidavit

I hereby declare that this bachelor thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

This thesis has not yet been presented to any examination authority, neither in this form nor in a modified version.

By signing this statement, I affirm that I have read the information notice on plagiarism, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Information notice on plagiarism:
http://www.ethz.ch/students/semester/plagiarism_s_en.pdf

Gianluca Vinzens Date

Contents

Chapter 1

Introduction

The internet consists of millions of servers, routers, and end nodes. Today, it is being extended by everyday objects to form the Internet of Things [1]. The idea is to be able to interconnect those physical artifacts and remotely access information about them and their environment. This concept finds application in many domains like businesses, factories, or interactive museums.

A concrete use-case of interest are smart household appliances, which are integrated into the Internet. A household offers a heterogeneous collection of devices with different technologies.
One approach is to interconnect these household appliances using the Constrained Application Protocol (CoAP) [2] for communication between them. We envision a dynamically changeable system of smart household appliances. All devices and sensors offer a RESTful interface in a Web-like infrastructure [3, 4] to access their data or trigger actions. One or more application servers are hosted on any of the more powerful appliances, such as a router. So called Apps [5] run on these servers, which interact with the devices and sensors. While the server is running, Apps can be installed, started, stopped, and restarted.

Californium [6], an application framework in Java, provides a CoAP library and a convenient API to realize applications for the Internet of Things. Applications using Californium export access through RESTful interfaces. HTTP-translation through an intercepting proxy is currently being implemented to extend Californium.[1] Another project covers the security aspects, which come with data exchanging systems.[2]
Actinium [7], an implementation of an app server, is an extension to Californium, which integrates Mozilla Rhino to provide support for JavaScript offering a convenient API, called CoAPRequest [8]. Apps exporting RESTful interfaces are

[1] http://www.vs.inf.ethz.ch/edu/abstract.html?file=/home/webvs/www/htdocs/edu/theses/mkovatsc-smart-city

[2] http://www.vs.inf.ethz.ch/edu/abstract.html?file=/home/webvs/www/htdocs/edu/theses/mkovatsc-secure-coap

hosted and run in Actinium.

Many applications for the Internet of Things perform similar reoccurring tasks [9]. We aim to find those tasks and split them up into reusable modules. This way, writing applications for the Internet of Things becomes more accessible and straightforward for the developers.
In chapter 2 we take a close look at already existing applications, which use sensors and actuators. Papers, amongst others on wireless sensor networks, serve as a basis to extract important requirements of applications on the Internet of Things. One of the more prominent papers is "Habitat Monitoring with Sensor Networks" [10], where thousands of sensors nodes are used to capture data on localized environmental conditions at the scale of individual organisms to learn more about the animals, plants, and people. "Shooter Localization in Urban Terrain" [11] is another often referenced paper, where audio sensors are used to localize a potential shooter in public places. The identified requirements are summarized in a requirements tree. It is then applied to find a classification of the provided applications into so called application classes. Finally in chapter 3 and 4, requirements often reappearing together in the application classes are implemented as reusable modules. They can then be combined to perform the more sophisticated routines found in the application classes. In order to evaluate the implemented modules, in chapter 5 we present an application kernel constructed from the modules for each application class. A detailed case study on a smart thermostat, which is a small collection of thermostat relevant applications, shows some of the application kernels in action.

Chapter 2

Classification

We aim to classify applications for the Internet of Things into so called application classes. A list of selected papers on wireless sensor networks and other distributed applications serve as a basis (Tables 2.1, 2.2 and 2.3). Through detailed study of the papers we first specify a requirements tree, which defines the most important characteristics of such applications. Applications covering the same requirements are then assembled to application classes.

Paper / Author	Application Description
Electronic Shepherd: a low-cost, low-bandwidth, wireless network system [12] Thorstensen B. et al. (2004)	The paper discusses a low-cost system that monitors flock behavior. It aids sheep and reindeer farmers to keep track of their animals during the grazing season.
Wireless Sensor Networks in Agriculture: Cattle Monitoring for Farming Industries [13] Kwong. K. H. et al. (2009)	Animal monitoring systems are used to continuously assess the condition of animals. The data is processed and reported to the farm manager.
Senslide: A Sensor Network Based Landslide Prediction System [14] Sheth A. et al. (2007)	Wireless Sensor Networks are used to detect landslide. Data is collected by periodically sampling sensor data and used for offline analysis. Rare events trigger an alarm.
Fence Monitoring: Experimental evaluation of a use case for Wireless Sensor Networks [15] Wittenburg G. et al. (2007)	Using a collection of sensors can improve the accuracy of event detection. Sensor nodes are attached to a fence to sense security relevant events, for example when people try to climb over.

Table 2.1: Papers discussing applications for the Internet of Things (1).

Paper / Author	Application Description
Shooter Localization in Urban Terrain [11] Maroti M. et al. (2004)	A network of acoustic sensors allows sniper detection in an urban environment. Using the measured time of arrival of sensing the acoustic shock wave the shooter can be located.
Fidelity and Yield in a Volcano Monitoring Sensor Network [16] Werner-Allen G. (2006)	Sensor Networks are used to monitor volcano actions. Data is collected and events such as earthquakes, eruptions, and other seismoacoustic events are detected.
Habitat Monitoring with Sensor Networks [10] Szewczyk R. (2004)	Thousands of sensor nodes are used to capture data on localized environmental conditions at the scale of individual organisms to learn more about the animals, plants, and people.
Not all Wireless Sensor Networks are Created Equal: A Comparative Study on Tunnels [17] Mottola L. (2010)	Densely-deployed light sensors are placed in tunnels to measure the light intensity. The sensed data is reported to a controller, which performs tuning of the illumination.
Weather Monitoring for Management of Water Resources [18] Hoogenboom G. (2001)	Strategically located stations are used to gather information about weather and climate to form an automated weather station network. The information is made accessible over the internet in a near real-time mode.
Timer Control for Telephone [19] Serby V. M. (1990)	A timed control mechanism is used to disable a telephone at preprogrammed times.
Timer Control for Television [20] Maclay W. R. (1986)	A timed control mechanism regulates the maximum time a television can be operated.
Periodic Control: A Frequency Domain Approach [21] Bittanti S. (1973)	The control of a plant is periodically varied. The optimal constant control can be improved by cycling.
Design and Implementation of a High-Fidelity AC Metering Network [22] X. Jiang et al. (2009)	AC energy usage data is collected and stored in a database. A web server is used to visualize the observed readings.

Table 2.2: Papers discussing applications for the Internet of Things (2).

Paper / Author	Application Description
System for Automatic Periodic Irrigation [23] D. D. Polizzi (1974)	Water irrigation is performed periodically with predetermined time intervals between the release cycles.
Wireless Sensor Networks for Personal Health Monitoring: Issues and an Implementation [24] A. Milenković et al. (2006)	Vital signs of patients are monitored using wearable sensors. The data is made accessible for feedback to maintain an optimal health status. When life-threatening changes are observed, medical personnel can be alerted.

Table 2.3: Papers discussing applications for the Internet of Things (3).

2.1 Requirements Tree

From the study of the papers we find a first coarse division of application routines into data collection, where data is requested from a source node, and actuation, where functions are actuated on a target node. We now take a closer look at both communication forms and identify a more detailed requirements tree (Figure 2.1). In addition, each application described in the considered literature is analyzed according to the requirements found (Table 2.4).

2.1.1 Data Collection

Data collection covers all communication forms, where data is read from a source. The state of the source node is not altered when gathering data. We distinguish data collection by **frequency**, **time range**, and the **sources** the data comes from.

Frequency

The frequency of data collection distinguishes between **low** and **high** sampling rates.

Low For monitoring systems like cattle monitoring or the electronic shepherd the data on the sources only change slowly or not very often. It suffices to read the data in a low rate and not miss important data changes.

High For many real-time applications like shooter localization and volcano monitoring the read data is analyzed for important events. Therefore the sam-

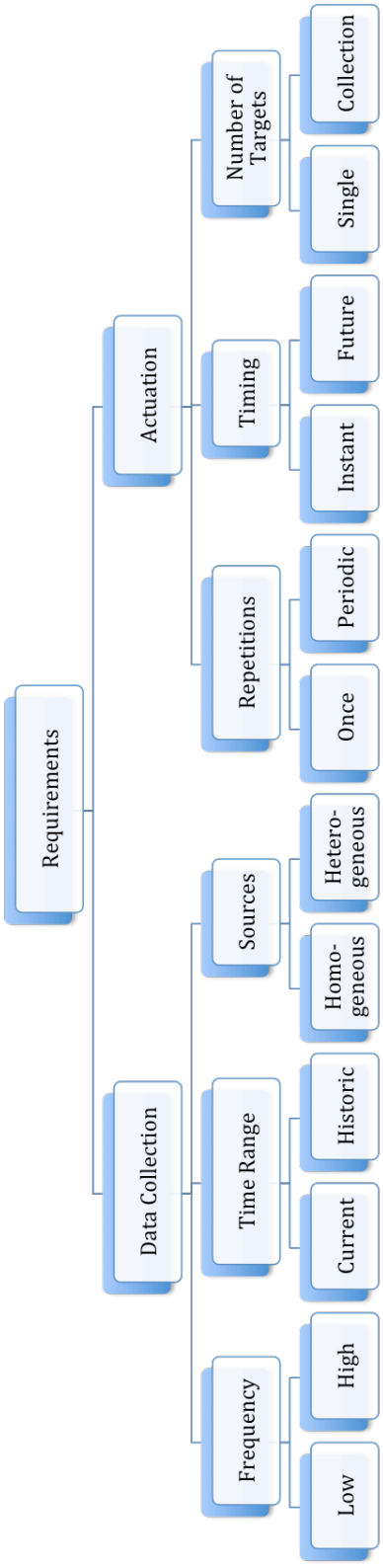

Figure 2.1: Requirements Tree: Hierarchical structure of requirements for applications of the Internet of Things.

pling rate needs to be high. Events need to be recognized instantly and actions are triggered to react on them.

Time Range

Data which is perceived by sensors is either **directly** processed or **stored in a database**.

Current Event detecting applications like fence monitoring or shooter localization use the current data to trigger actions when noticing interesting changes.

Historic Often, gathered data is stored in a database to be made accessible later on. When monitoring a volcano or a habitat over a longer period of time historic data is essential for analysis. The historic data can then be made available with time constraints, which take into account the time the data was sensed. It is also possible to retrieve aggregated representations of the data directly from databases.

Sources

Usually, monitoring applications use many sensors to gather data. We distinguish between applications that use **homogeneous** sensors and **heterogeneous** sensors.

Homogeneous When all sensors are of the same type and record the same kind of data, we talk about homogenous sensors. Usually each sensor is responsible for a separate entity, as in cattle monitoring, or the sensors are geographically distributed to cover a larger area. The latter is the case for the shooter localization application, where they make use of the variation of the arrival time of the sound at the sensors, depending on the location of the sensor. It is also possible, that different sensors work at various sampling rate frequencies, to reach large or even full area coverage, but limiting data traffic.

Heterogeneous In order to capture different aspects of an environment, applications like volcano monitoring or weather monitoring use a whole range of different sensors. The sampled data is then analyzed independently or combined and aggregated to more complex and meaningful data.

2.1.2 Actuation

Communication forms, where data is sent to a target node are covered by the actions. Actions trigger events on the target or change its state. We distinguish

actions by the **repetitions**, the **timing**, and the **number of targets**.

Repetitions

Data can be sent different numbers of times, **once** or **periodically**, to a target.

Once In the usual case, the data is sent once to a target, without any repetitions. This may be in form of an alarm because of a perceived event of a monitoring application like volcano monitoring. It can also be a reaction to a change in a state as for the tunnel application controlling the lights.

Periodic When a sequence of actions follows an observable pattern we talk about periodic actions. Plants can be run or water-irrigation systems can be started using periodic control. Both the interval and the data sent with the action may differ, as long as it follows a pattern. The duration of periodic actions is finite, when the number of repetitions is known at the beginning, or infinite, where only a forced shut down can stop the periodic execution.

Timing

An action can be executed **instantly** or be timed for a **future** point in time.

Instant The more common case is, when an action is instantly executed. Usually after a specific event was detected, as seen in shooter localization or fence monitoring, an alarm is issued immediately.

Future Sometimes, actions need to be timed to trigger in the future. The execution time of the action is already known beforehand though. Telephones or televisions can be instructed to turn on or off at a predefined time.

Number of Targets

A single action can target a **single** node or a **collection** of nodes.

Single In the general case, an action is only intended for a single target node. Alarms for monitoring applications are triggered when an event is recognized and displayed in some form.

Collection An action can control a whole group of nodes, for example when the lights are regulated in tunnels.

	Data Collection			Actuation		
Paper	Frequency	Time Range	Sources	Repetitions	Timing	Number of Targets
Electronic Shepherd	Low	Historic	Homogeneous	Once	Instant	Single
Cattle Monitoring	Low	Historic	Homogeneous	Once	Instant	Single
SenSlide	Low	Historic	Homogeneous	Once	Instant	Single
Fence Monitoring	High	Current	Homogeneous	Once	Instant	Collection
Shooter Localization	High	Current	Homogeneous	Once	Instant	Single
Volcano Monitoring	High	Historic	Heterogeneous	Once	Instant	Single
Habitat Monitoring	Low	Historic	Heterogeneous	Once	Instant	Single
Study on Tunnels	Low	Current	Homogeneous	Once	Instant	Collection
Periodic Control of Plants	-	-	-	Periodic	Future	Collection
Weather Monitoring	Low	Historic	Heterogeneous	-	-	-
Timer Control for Telephone	-	-	-	Single	Future	Single
Timer Control for Television	-	-	-	Single	Future	Single
AC Metering	High	Historic	Homogeneous	-	-	-
Periodic Irrigation	-	-	-	Periodic	Future	Collection
Health Monitoring	Low	Historic	Heterogeneous	Once	Instant	Single

Table 2.4: Requirements for each application from the literature.

2.2 Application Classes

In a next step we identify application classes from the inspected list of applications. Each application described in the papers is analyzed according to the requirements. Applications with similar requirements are then summarized within a single application class (Figure 2.5).

2.2.1 Data Collection and Visualization

A first application class is "Data Collection and Visualization". Different sensors, homogeneous or heterogeneous, generate data, which is read at low or high frequency and stored in a database. Queries using time dependent constraints and aggregation provide different views of the sensed historic data. No data is being processed to find special events and trigger actions. The *weather monitoring application* or *AC monitoring application* are representatives of this class.

2.2.2 Environmental Event Monitoring

In a second application class, "Environmental Event Monitoring", data is perceived by sensors. The sensors may be homogeneous or heterogeneous and data is read at low or high frequency. Aggregation and processing mechanisms automatically analyze the data to recognize important events. Those events then immediately trigger actions, e.g. in form of alarms. The generated data from the sensors is stored in a database. Database data is incorporated into the event analyzing process or just used to create a history of the environmental state. Monitoring applications like *volcano monitoring* or *cattle monitoring* belong to this class, which not only gather data, but also recognize events and trigger actions.

2.2.3 Control System

The third application class, "Control System", constantly reads new data, at low or high frequency, from some environment sensor nodes. The data captured by homogeneous or heterogeneous sensors is analyzed and processed immediately. Changes in the state of the environment instantly trigger actions, which react to the changes. The data is not stored in any database. The sole purpose of the application is to give real-time response on the state of the environment and act as a regulating mechanism. A security application like *fence monitoring* or a control application as described in the *study on lights in tunnels* apply to this class.

Application Class	Paper / Application
Data Collection and Visualization	Weather Monitoring AC Monitoring
Environmental Event Monitoring	Electronic Shepherd Cattle Monitoring Senslide Volcano Monitoring Habitat Monitoring Health Monitoring
Control System	Fence Monitoring Shooter Localization Study on Tunnels
Timed Control	Timer Control for Telephone Timer Control for Television
Periodic Control	Periodic Control of Plants Periodic Irrigation

Table 2.5: Assignment of each application to the corresponding application class.

2.2.4 Timed Control

A fourth application class, "Timed Control", defines an action, which is executed at a future point in time. This action may target a single node or a collection of nodes. The collection of targets can depend on the data sent with the action. An interface is used to provide the execution time and data for the action. *Timer control applications for telephone and television* belong to this class.

2.2.5 Periodic Control

A last application class, "Periodic Control", performs actions according to a predefined pattern. This control mechanism may target a single node or a collection of nodes, depending on the data sent with the action. The periodic action is usually triggered through an interface. This class is represented by the *periodic control of plants application* or the *water irrigation system.*

Chapter 3

Module Design

The created requirements tree from the previous chapter serves as basis for the selection of the modules. The goal is to cover all requirements aspects with at least one module each. We take the knowledge gained from the applications discussed in the papers and have each module fulfill requirements that most of the time appear together. The modules can later on be interconnected through their interfaces to form more complex mechanisms.

All modules should follow a unified RESTful API design. A consistent resource layout makes them more accessible and intuitive. Each module holds several module tasks, which represent individual instances of the module's functionality. This way module tasks can be bundled in a single module entity and the system remains more well-arranged. Initially only one resource, the **/tasks** resource, is available (Figure 3.1). New module tasks are then created inside that resource (Figure 3.2).

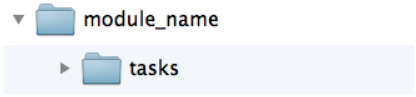

Figure 3.1: Initial setup for all modules.

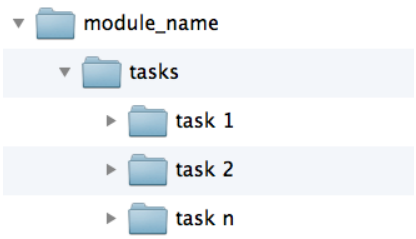

Figure 3.2: Tasks are created inside the tasks resource.

3.1 Persisting Service

A first module, Persisting Service, performs two main jobs. It is responsible for reading data and storing it in a database (Figure 3.3). On the other hand it also

offers a collection of accessing mechanisms to the database and its data for other applications (Figure 3.4).
For historic data to be available at any time, the data has to be stored in a database. This module requests the data from a source either through polling, or preferably if available using an observer mechanism. Checking for the availability of such an observer mechanism is the responsibility of this module. Incoming data is then stored in the database. For each data value the source node and the time of arrival is stored with it.

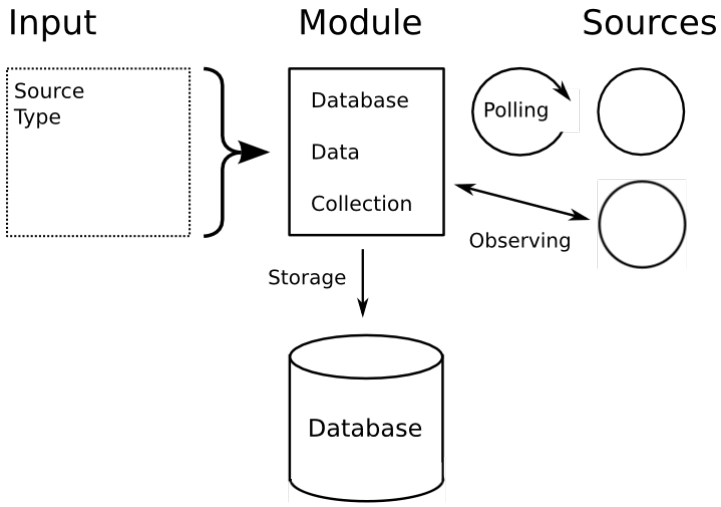

Figure 3.3: Persisting Service Module Data Collection.

The data stored inside the database has to be made accessible for other nodes of the system. A data retrieval request on the database is always dependent on the source of the data. Timing constraints such as since, a time range, the last number of values stored or the newest can be selected by each retrieval request. On top of that, the sum, the average, the maximum, or the minimum can be selected as aggregates if suitable for the data type. Obviously it is possible to combine the two forms of data retrieval, that is to specify a timing constraint and an aggregation at the same time.
The persisting service can be instructed to store data for a specific device resource. For each such source resource, a persisting task has to be created. A persisting task can be instructed to store number values or string values. In theory, numbers could also be stored as strings, but a number instance offers more data retrieval options in form of aggregation.
When the persisting task is initialized, the persisting service does not start collecting data right away. It has to be started by sending a PUT request onto the **/running** resource. Data stored in the database can be retrieved in both states, running or not running.

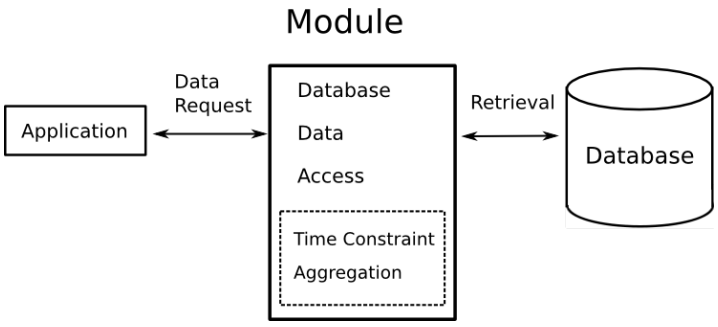

Figure 3.4: Persisting Service Module Database Access.

The **/history** resource encompasses all the retrieval resources. Persisting tasks storing string values can only retrieve data with a time dependent constraint. When numbers are stored, the user also has the option to specify an aggregation function for most data retrievals.
Data is requested by sending GET requests to the desired history resource. When necessary, the required options, such as a date, need to be passed together with each request. As default, only a list of sole values is returned for a request. When desired, time dependent data retrieval can also return the storage date together with each data value. Each data retrieval resource is observable. This way, any other device can register as observer and is notified about new data received and stored by the persisting task.

3.2 Timed Action

Timed actions can only be PUT, POST, or DELETE requests. In order to initialize a timed action task in this module (Figure 3.5), the execution time, the target resource and the operation need to be specified. The time of execution can either be defined as a complete date with time, a date only, a time only, or a delay. We use the conventions, that for a single date the action executes at midnight and for a single time the current date is chosen. The timed action may also send information in form of a request payload, which is also defined at creation.
The **/datetime** resource returns the intended time for executing the request. Both the **/datetime** and the **/payload** resource can be changed through a PUT request, as long as the datetime is not reached yet. As soon as the timed action is executed, the task deletes itself.

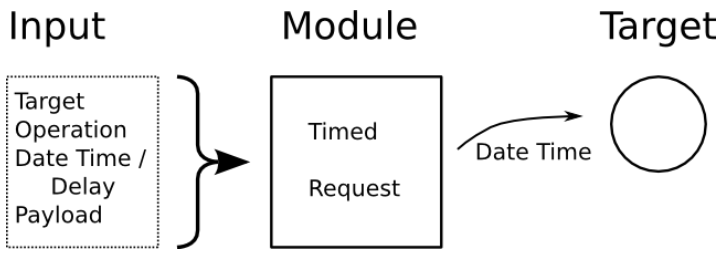

Figure 3.5: Timed Request Module.

3.3 Periodic Action

Periodic actions can only be PUT, POST, or DELETE requests. This module (Figure 3.6) makes it possible to create a periodic action task for a target node with a predefined period. The payload can also be specified at creation. Both the period and the payload can be changed while the periodic action task is running. The changes take effect the next time the period is run out and starts over again.
Period and payload may change over time in a predefined pattern. Instead of specifying a constant period or payload, it is possible to define a function, which returns the changed values according to the observed pattern. Some of these functions are preimplemented in the module and can simply be called. For a wider range of specifications the user may also write his or her own pattern functions. Whenever both, the payload function and the payload, are defined, the function is active, and the constant payload is ignored. The same holds for the period.

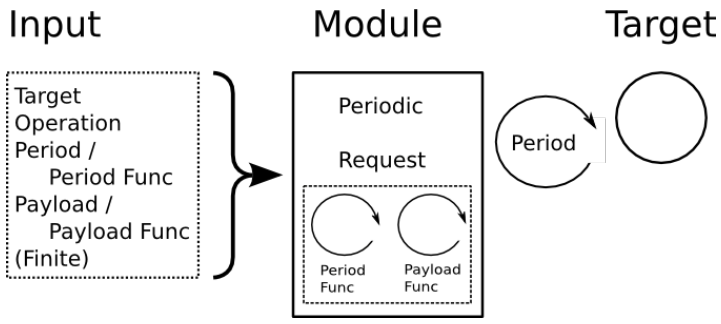

Figure 3.6: Periodic Request Module.

By default, all periodic actions are infinite. In case of a finite periodic action, the user selects a finite number of repetitions at initialization of the task. If this is the case, a resource called **/finite** returns the number of repetitions, while another resource, **/remaining**, returns the remaining repetitions. After finishing the

fixed number of repetitions the periodic action task remains accessible and can be restarted if necessary.
A running task can be stopped at any time. When needed, the periodic action task can either be restarted or continued where it left off.

3.4 Multicast

A single request can be intended for a collection recipients. The multicast module (Figure 3.7) can perform PUT, POST, or DELETE multicasts. It basically multiplies the request, while changing the target node for each.

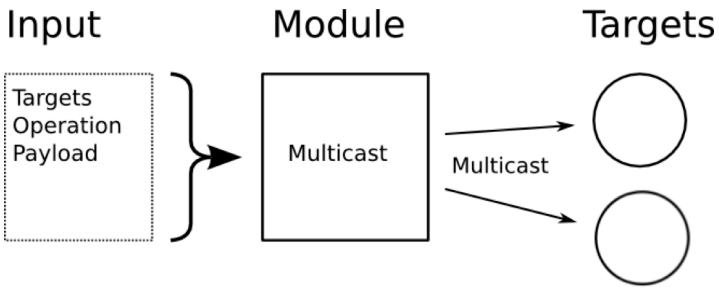

Figure 3.7: Multicast Module.

For a repeated multicast, the user defines a separate task for it. The information of all the target nodes needs to be provided when initializing a new task. In addition, for each target a decision function (Table 3.2) can be defined, which checks the payload for some characteristic. Whenever a PUT, POST or DELETE request is sent to the **/cast** resource of such a task, it multiplies the request and sends it to all specified targets, where the decision function returns true. The payload is automatically passed with the multicast.
For a single multicast, no multicast task has to be created. The module offers the option of multiplying a request on the fly by sending any PUT, POST or DELETE request to the **/single** resource. In this case the targets information and payload need to be passed with the request.

3.5 Multiple Aggregates

Sometimes, it is necessary to collect data from a collection of sources and then optionally use a aggregation mechanism to combine the values. When initializing a multiple aggregate task for this module (Figure 3.8), the source nodes are specified and the module starts observing all of them.
For the aggregation of the incoming values, the module implements a collection

Decision Function	Description
Equal	Checks the incoming value for equality with a specified value.
Not Equal	Checks the incoming value for inequality with a specified value.
Greater	Checks the incoming value to be greater than a specified value.
Greater Equal	Checks the incoming value to be greater than or equal to a specified value.
Less	Checks the incoming value to be less than a specified value.
Less Equal	Checks the incoming value to be less than or equal to a specified value.
Contains	Checks the incoming value to contains a specified value.
Prefix	Checks the incoming value to start with a specified value.
Suffix	Checks the incoming value to end with a specified value.
Own	The user defines his or her own function to check the incoming value for a property.

Table 3.2: Decision Functions.

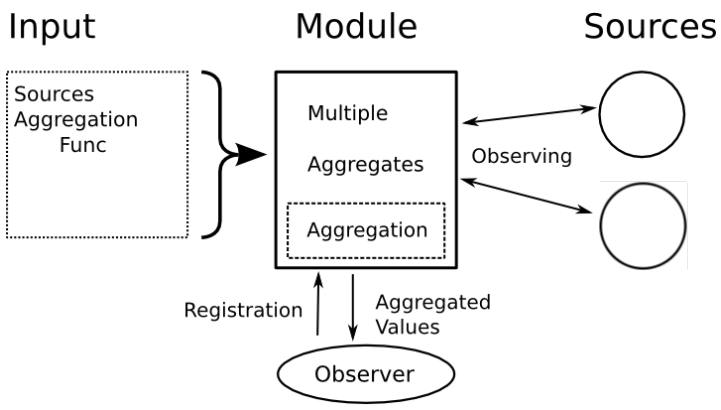

Figure 3.8: Multiple Aggregates Module

Modification/ Aggregation Function	Description
Sum	Computes and updates the sum of all incoming values.
Average	Computes and updates the average of all incoming values.
Maximum	Computes and updates the maximum of all incoming values.
Minimum	Computes and updates the minimum of all incoming values.
Add	Add a specified value to the incoming value.
Subtract	Subtracts a specified value to the incoming value.
Multiply	Multiplies a specified value to the incoming value.
Divide	Divides the incoming value by a specified value.
Modulo	Computes the incoming value modulo a specified value.
Prefix	Prepends a specified value to the incoming value.
Suffix	Appends a specified value to the incoming value.
Own	The user defines his or her own function to modify the incoming value.
Newest / None	The value is passed without change.

Table 3.4: Modification / Aggregation Functions

of predefined aggregation functions (Table 3.4). The values can simply be passed along without change or be aggregated or modified using a more complex function. Own aggregation functions can also be written by the user and passed to the module. The aggregation function has access to both the value and the source node the value belongs to. While the multiple aggregate task is running, the aggregation function can be changed at any time. Temporary information of an aggregate functions are lost in doing so.
The task itself offers an observing mechanism for others to register for the aggregated values. Any application can register as observer on the **/aggregate** resource. Whenever a new value is received by the task, the aggregated value is passed to all observers. An observer may choose, whether the sole value or the value with its device information is requested. The device information is a simple integer corresponding to the device number specified at creation.

3.6 Control Loop

The control loop module (Figure 5.3) reads data from a source node, modifies the data or creates new data and sends it to a target node. When a new control loop task is created, it registers at the source and waits for new incoming values.

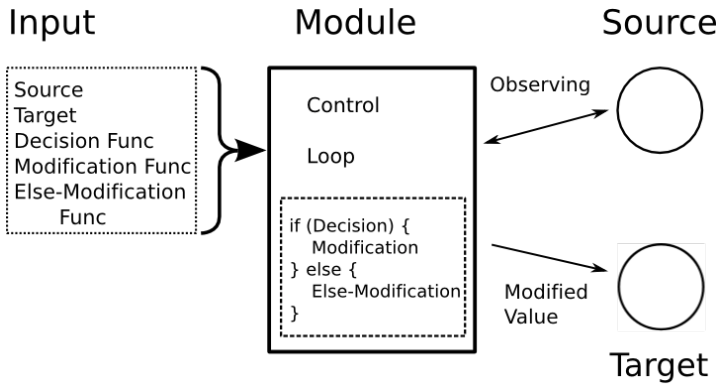

Figure 3.9: Control Loop Module.

For the control loop altering mechanism, a set of predefined functions are available. It is possible to only specify a modification function (**/modifyfunc** resource), which processes all received values from the source (Table 3.4). If only a subset of all incoming data values should be altered a decision function (**/decisionfunc** resource) can be selected (Table 3.2). Only if the decision function returns true, the value will be processed. A third option is to also process the incoming value when the decision function returns false. In this case, two modification functions need to be defined, one for when the decision function returns true, the other for when it returns false (**/modifiyfuncelse** resource). The module has the most basic decision and modification functions preimplemented, which can simply be selected by the user. To give the user full flexibility in designing the control loop, the decision function and modification functions can also be written by the user and passed to the module. If necessary, both the decision function and the modification functions can be changed at runtime, which may involve loss of temporary information.

3.7 Push Simulation

The push simulation module (Figure 3.10) is different from all the others. It is not designed to fulfill a specific requirement from the requirements tree defined in the last chapter as all the others do. It merely aids the programmer wherever no observing mechanism is available.

Observing a source node is preferable over polling. Therefore, all designed modules always observe the sources. If the observer mechanism is not implemented on a source, this module acts as an intermediate entity, which simulates the observing mechanism through polling. Data is continuously read from the source node and only passed to its observers when the value changes. The idea is, that the device missing the observer mechanism is chosen as the source. Any application can now register as observer at the **/value** resource and be notified, when new values are received.

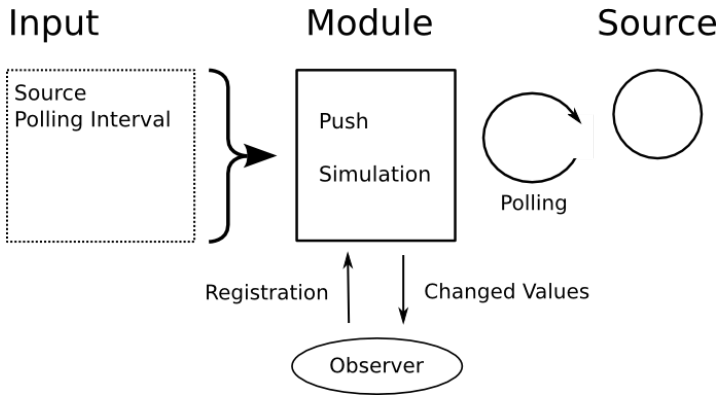

Figure 3.10: Push Simulation Module.

At initialization, the source node is specified. The polling period can be passed at creation or is set to a default value. The default value for the polling interval is 1000ms. It can be changed while the task is running through a PUT request on the **/poll** resource. The change of the polling interval takes effect once the current polling iteration ends and the next one takes over.

Chapter 4

Implementation

In this chapter we go more into detail on how the modules were implemented. We take a closer look at the different components and frameworks we chose. In order to make the modules more accessible, a web frontend was implemented, which provides a more user-friendly interface to interact with each module. For the modules to be successful in larger systems, the execution times need to be performant. A small performance evaluation calculates those times and provides some statistical information.

4.1 RESTful Communication

For the Internet of Things, everyday objects are being interconnected in a Web-like infrastructure. Devices and sensors export their functionality through RESTful interfaces, through which data can be retrieved and devices can be controlled. The Constrained Application Protocol (CoAP) is used as communication protocol to connect the devices and sensors. It is a RESTful protocol similar to HTTP. Instead of TCP it uses UDP as network protocol.
Californium is a Java framework of CoAP. Complex resources can be programmed in Java and profit from the strengths Java contributes. Each resource is implemented as a single Java class, where different code is provided for GET, PUT, POST, and DELETE requests.

4.2 Module Implementation

The modules can be implemented in Java or JavaScript, depending on their complexity. The persisting service performing a lot of I/O operations is considered to be a heavy weight module. It is therefore implemented in Java using the Californium framework. CouchDB [1] [2] is chosen as database, mostly because it

[1] http://couchdb.apache.org/
[2] http://guide.couchdb.org/index.html

is well suited for distributed systems. Ektorp [3] [4] [5] serves as a link from Java to CouchDB. All other modules are implemented in JavaScript as apps running in Actinium.

4.2.1 CouchDB

As a database for the persisting service we propose the use of CouchDB. CouchDB is an open source project that provides a noSQL database system. NoSQL databases do not know about schemas such as tables like traditional databases do, but follow a document based approach. It is neither a relational nor an object oriented database.
CouchDB stores its content in a collection of semi-structured JSON documents. Each document keeps a unique ID and a revision number. Different data types such as text, number, boolean, or list are possible.
The document collection can then be structured using JavaScript views. Through view representations the data is filtered, organized and reported. A view is just a special document inside the database executing a map-reduce function over all other documents. The map function filters the documents and returns a new set of documents (Figure A.1). The reduce function can then serve as an aggregation mechanism to perform computations over the documents returned by the map function. To increase caching efficiency, the documents are passed in different-sized blocks to the reduce function. The reduce function first only reduces those blocks, each for itself. The resulting documents are then again passed to the reduce function to be rereduced (Figure A.2).
CouchDB has a RESTful API, which can be accessed using HTTP requests.

Advantages

Because no schema is involved for a CouchDB database, it is highly scalable. New document types can be added without having to change anything inside the database.
Replication of the database is very simplified. Replicas can be created, which provide full database interactivity. CouchDB uses an incremental replication approach. In order to replicate the design of the database, no further measures have to be taken. View representations are specialized documents and therefore automatically replicated with the database.
CouchDB can be accessed just like a web server. It offers a RESTful API and can be accessed using HTTP requests.
JavaScript in combination with JSON objects are used to store the data and filter it. Both are very commonly used in distributed systems.

[3] https://github.com/helun/Ektorp
[4] http://www.ektorp.org/reference_documentation.html
[5] http://ektorp.org/tutorial.html

Disadvantages

CouchDB is a document based database where view representations simulate tables and other schemas. To make data retrieval efficient, views build indexes on top of the documents. This setup forbids passing parameters to the map-reduce function. Documents can only be filtered by their key, generated by the map function of a view.

Ektorp

The persistence service is written in Java. Ektorp, a persistence API for Java using CouchDB as storage, serves as a link between the service and the database. Most of the CouchDB API is covered by Ektorp.

4.2.2 Actinium

Actinium is an extension of the CoAP framework Californium. Through integration of Mozilla Rhino it provides support for JavaScript apps. All apps implement a RESTful resource tree. Each resource can react to GET, PUT, POST, and DELETE requests.
Actinium provides the possibility to install new apps while the App Server is running. Apps can be started, stopped and restarted at runtime. This mechanism supports a much more dynamic system setup than in stand-alone Java Applications utilizing the Californium framework.

4.3 Web Frontend

To make interaction with the modules more user-friendly a web frontend was designed. Each module can be controlled from within the frontend. New tasks can be created or altered with just a mouse click. No knowledge of the actual RESTful interface of the modules is required (Section A.2).
The web frontend communicates using HTTP. All modules run in Actinium offering a RESTful CoAP interface. A proxy [6] translates HTTP requests to CoAP requests. All requests to the apps are first sent to the proxy, which redirects them to their actual target.

[6] http://www.vs.inf.ethz.ch/edu/abstract.html?file=/home/webvs/www/htdocs/edu/theses/mkovatsc-smart-city

4.4 Module Performance

The modules need to be tested for their performance. In a complex household system with many devices and several different applications controlling them, it is important, that the communication is reliably efficient.
We built a small testing suite, which iteratively sends data through the modules. For each module source resources, target resources, and, if necessary, observing resources are created. The time is first stored at the source, before the data is sent to the module. The module processes the data and sends it off to the target or the observer, where the time difference is recorded.
The tests were performed on a Mac OS X, 2.4 GHz Intel Core i7 and 4 GB RAM. Each module was tested separately one after the other, executing 1000 iterations of sending data and calculating the time. The data is presented in a line chart where the number of occurrences for each time in milliseconds is displayed. Additionally, statistical measures such as the average, standard deviation, maximum and minimum are provided for deeper understanding of the data. No other apps were actively running during the testing phase.

Statistics			
Average	32.16 ms	Maximum	433 ms
Standard Deviation	71.21 ms	Minimum	5 ms

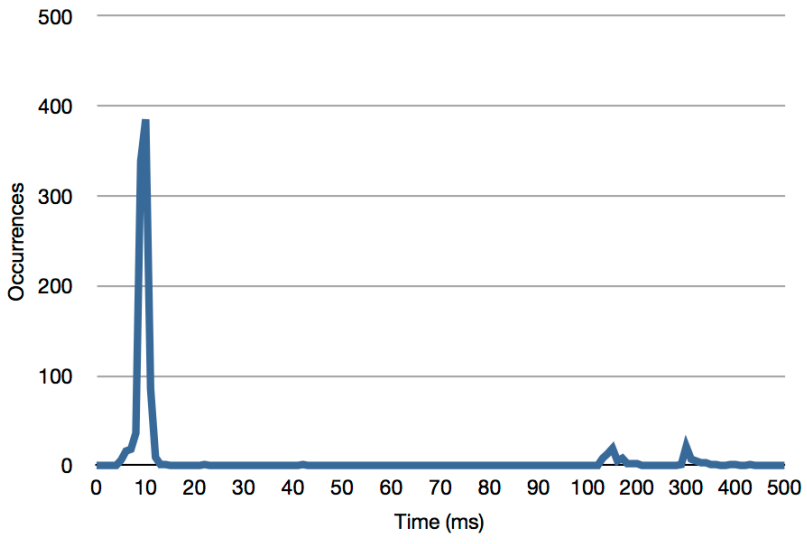

Figure 4.1: Persisting Service: Statistics and Execution Time Line Chart.

4.4.1 Persisting Service

For the persisting service, the testing suite creates a source resource and an observing resource. A new persisting service task registers as observer on the source. Meanwhile the observer resource registers as observer on the the persisting service task to receive the newest incoming values. Periodically, the source is triggered to change the value and pass it to the persisting service task, which stores the value and sends it to the observer resource. The time between the source and the observer is recorded (Figure 4.1).
Since the observer is only interested in the newest value stored in the database, the persisting service only has to fetch one document from it. For requests of multiple values or aggregation, the time would probably increase.

4.4.2 Timed Action

To test the timed action module the testing suite creates a target resource. Repeatedly, a new timed action task is created. The intended time to trigger is stored. When the timed action arrives at the target, the difference between triggering the request and receiving the request is computed (Figure 4.2).

Statistics			
Average	3.83 ms	Maximum	27 ms
Standard Deviation	2.17 ms	Minimum	0 ms

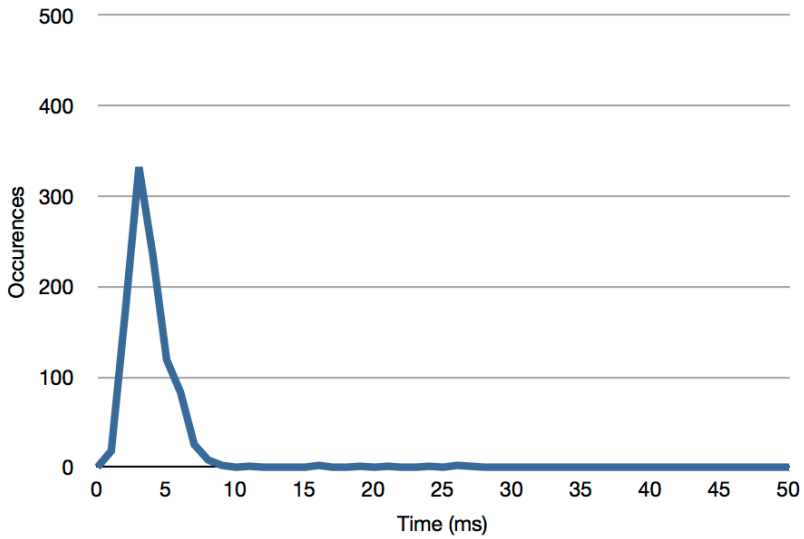

Figure 4.2: Timed Action: Statistics and Execution Time Line Chart.

4.4.3 Periodic Action

The testing suite creates a target resource to test the periodic action module. A periodic action task with a constant period is initialized. To compute the time for the periodic action to send the requests repeatedly, the deviation of the strict periodic interval is computed when the request arrives at the target. The change in deviation corresponds to the delay caused by the periodic action module (Figure 4.3).

Statistics			
Average	1.09 ms	Maximum	13 ms
Standard Deviation	1.01 ms	Minimum	0 ms

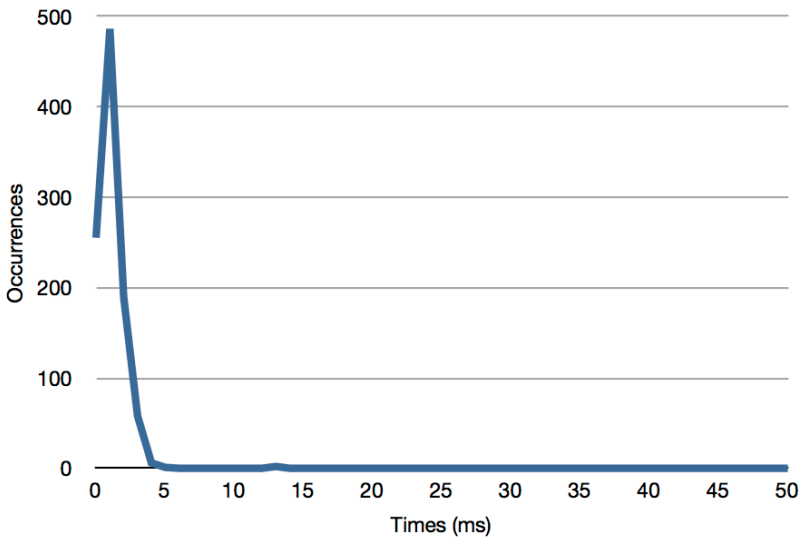

Figure 4.3: Periodic Action: Statistics and Execution Time Line Chart.

4.4.4 Multicast

The testing suite creates two target resources to test the multicast module. To multiply a request on the created targets, a multicast task is initialized. Periodically, a request is sent to the task and the time of sending it is stored. As soon as the request arrives at the last target, the time used to pass through the module is recorded (Figure 4.4).

Statistics			
Average	6.21 ms	Maximum	601 ms
Standard Deviation	23.62 ms	Minimum	2 ms

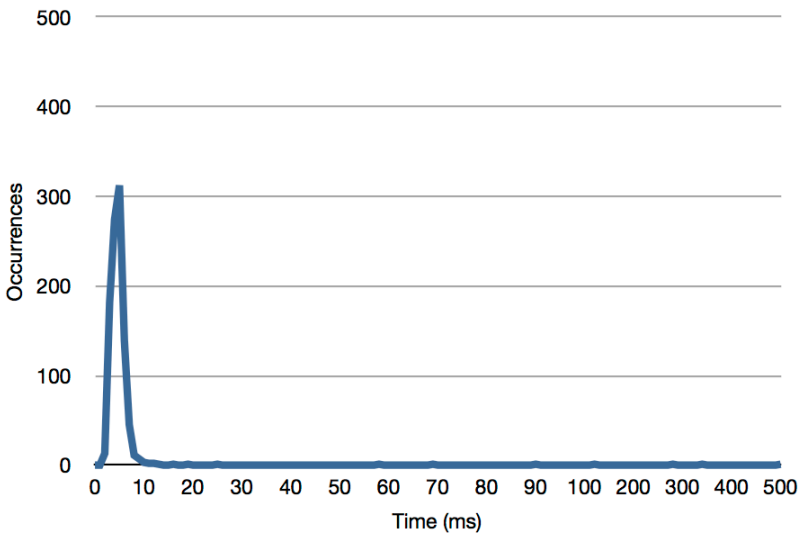

Figure 4.4: Multicast: Statistics and Execution Time Line Chart

4.4.5 Multiple Aggregate

The multiple aggregate module is tested by creating two source resources and an observer resource in the testing suite. A multiple aggregate task is initialized, which starts observing the sources. The observer resource registers as observer on this created task. Repeatedly the sources are triggered to change their value, send it to the multiple aggregate task, and store the sending time. The task processes the data and sends it to the observer. Whenever a value arrives at the observer, the time is recorded (Figure 4.5).

Statistics			
Average	1.72 ms	Maximum	57 ms
Standard Deviation	3.16 ms	Minimum	0 ms

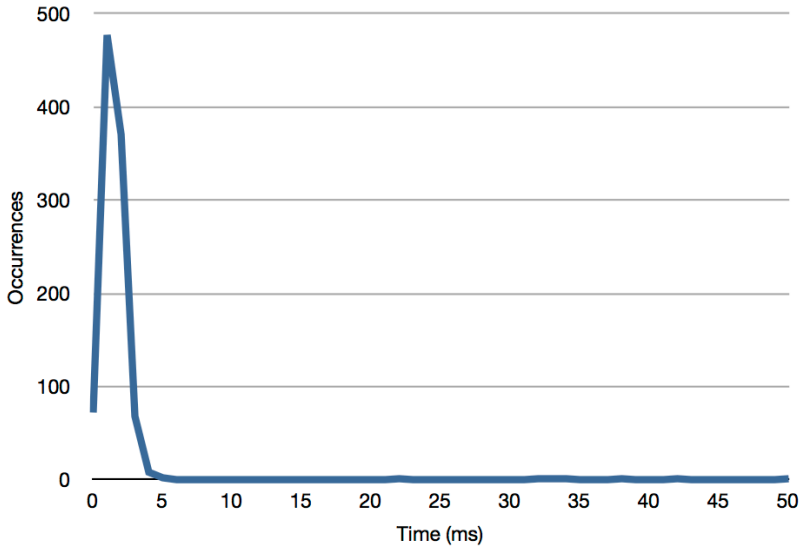

Figure 4.5: Multiple Aggregate: Statistics and Execution Time Line Chart

4.4.6 Control Loop

For the control loop module the testing suite creates both a source resource and a target resource. A new control loop task is initialized, which registers as observer on the source. The source is triggered periodically to change the value and send it to the control loop task. When the source sends the data to the control loop, it saves the sending time. The data is processed and passed to the target, where the time is stopped and recorded (Figure 4.6).

Statistics			
Average	3.66 ms	Maximum	26 ms
Standard Deviation	1.63 ms	Minimum	0 ms

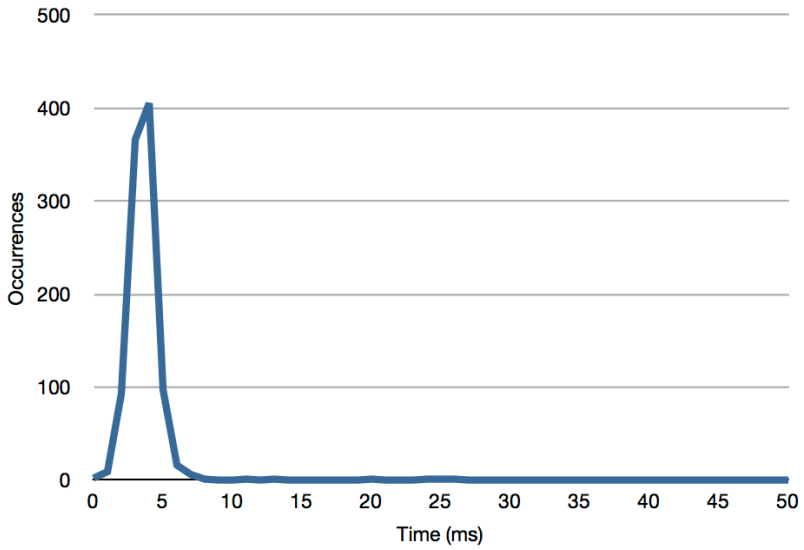

Figure 4.6: Control Loop: Statistics and Execution Time Line Chart.

4.4.7 Push Simulation

For the push simulation module the testing suite creates a source resource and an observer resource. A push simulation task is initialized, which polls on the source. The observer resource registers as observer on the push simulation task. When the data changes on the push simulation, it sends the new value to the observer. The time from the source to the observer is recorded (Figure 4.7).

Statistics			
Average	1.82 ms	Maximum	345 ms
Standard Deviation	13.41 ms	Minimum	1 ms

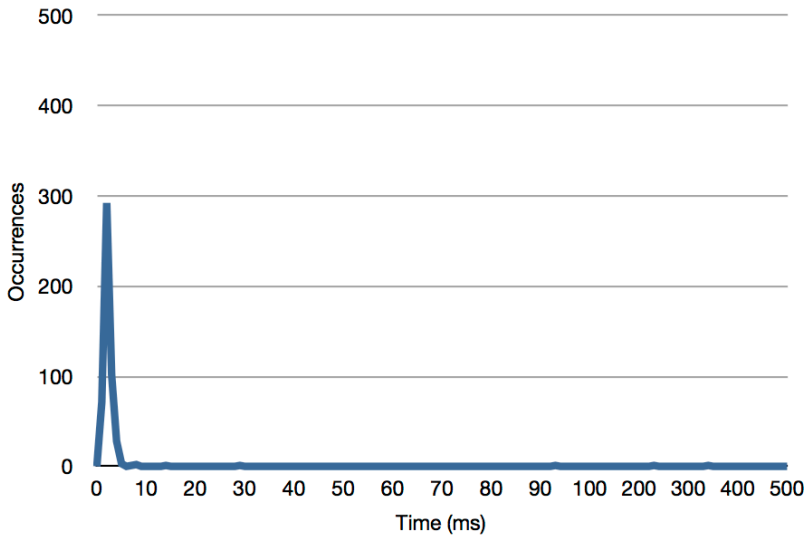

Figure 4.7: Push Simulation: Statistics and Execution Time Line Chart

Chapter 5

Evaluation

In a last step we evaluate the usefulness of the implemented modules. The goal was to create reusable modules for the specified application classes. In order to evaluate the modules, we build an application kernel for each application class by interconnecting the modules in different ways. The application kernel should perform the core functionalities defined by each application class.
To show some of the application kernels in action and to test their functionalities, we conduct a detailed case study. A small test application called **smart thermostat**, which is a collection of thermostat relevant applications, controls existing thermostats utilizing the yahoo forecast information [1] [2]. The data is visualized using the web frontend.

5.1 Application Kernels

Each application class should be realized using a subset of all modules (Table 5.1). These interconnected modules build so called application kernels, which perform most of the core functionalities described in the application class.
To draw a bow back to the smart household appliances, for each application kernel we point out an example of a smart home application. All these application representatives are discussed in the observed literature, which also served as basis for the application classes.

5.1.1 Data Collection and Visualization

The data collection and visualization application class stores data in a database and makes it visible through a display. It can be realized using the **persisting service** only. Both homogeneous and heterogeneous sources are being observed by the persisting service. Displaying applications can retrieve the data directly

[1] http://developer.yahoo.com/weather
[2] http://weather.yahooapis.com/forecastjson?w=2502265

Application Class	Persisting Service	Timed Request	Periodic Request	Multicast	Multiple Aggregate	Control Loop
Data Collection and Visualization	X					
Environmental Event Monitoring	X				X	X
Control System				X	X	X
Timed Control		X		X		
Periodic Control			X	X		

Table 5.1: Modules used for each application class found in the analyzed literature.

from the persisting service module. Data can be retrieved using time constraints or aggregation if suitable. (Figure 5.1)
The AC metering application is a representative of household applications, which can be realized using this kernel. The persisting service collects all the AC data from sensors and stores it in a database. Web services or home displays can then access the data, optionally in aggregated form, to visualize it.

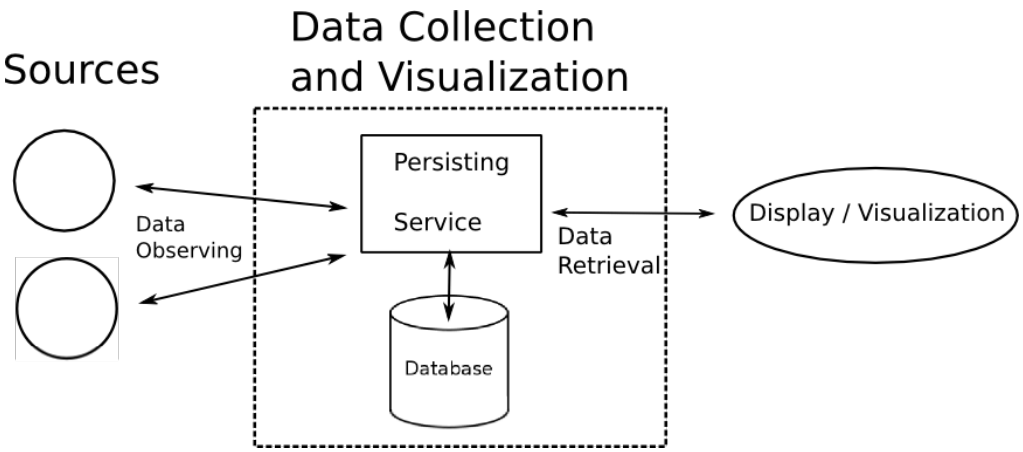

Figure 5.1: Application Kernel Data Collection and Visualization.

5.1.2 Environmental Event Monitoring

The environmental event monitoring application class observes a collection of sources. If an event occurs, a special action is triggered. All the data is stored in a database for later retrieval. The **multiple aggregate** module observes the sources. The incoming values can be aggregated if needed and are then passed to the **control loop** module. This is were the data is analyzed and possibly an event is recognized and a corresponding action produced. All the data from the sources, the aggregated value and the control loop output are stored in a database using the **persisting service** module. This way, the data history is always accessible for later analysis. (Figure 5.2)
To give an example from smart household applications, we take a closer look at the health monitoring application. Wearable sensors provide data, which is collected by the persisting service. Using the multiple aggregate module, the gathered data can already be aggregated before it is being stored. Through combination of aggregation of the data and analyzing it using the control loop module, life-threatening situations can be detected and health personnel can be alerted.

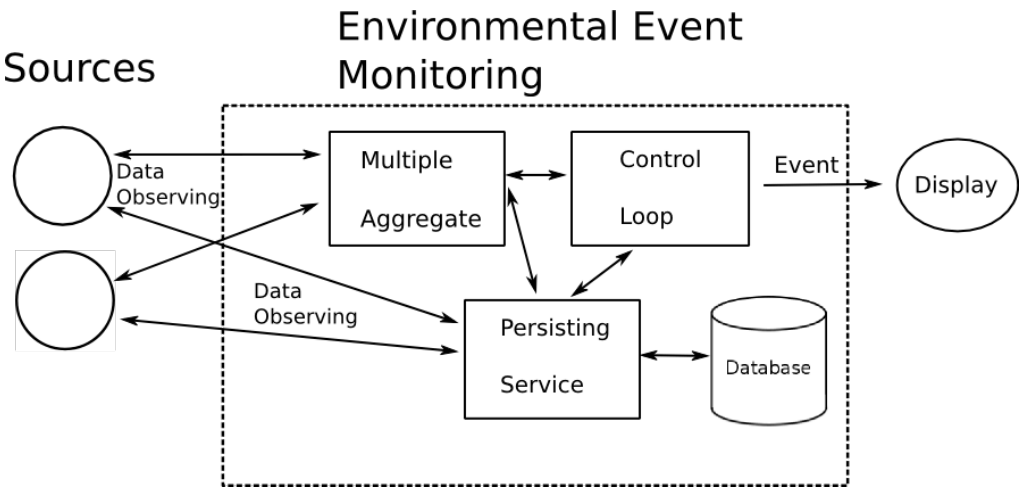

Figure 5.2: Application Kernel Environmental Event Monitoring.

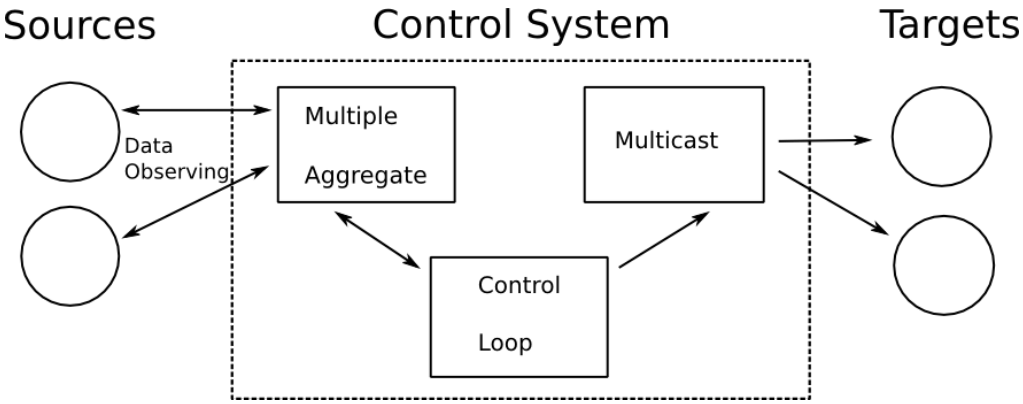

Figure 5.3: Application Kernel Control System.

5.1.3 Control System

A control system observes source data and triggers actions on targets depending on the incoming values. This application class can be obtained by having the multiple aggregate module observe the sources. The produced aggregated value is passed to the **control loop** module, where it triggers an action. The action can be multiplied for a collection of targets using the **multicast** module. (Figure 5.3
To make smart homes saver from intrusion of burglars, the fence monitoring application can detect illegal trespassing. Many sensors along the fence observe the environment. The data is collected and aggregated by the multiple aggregate module and passed to the control loop module for analyzation. Whenever an intruder is detected, an alarm can be sent to several targets. These targets could be displays in the house, mobile phones of the owners, or even directly the police.

5.1.4 Timed Control

For the timed control application class the user provides data and the date and time a specific action should be triggered. The input information is passed to the **timed action** module. As soon as the execution time is reached, the specified action is activated. The request can be multiplied for several targets using the **multicast** module. (Figure 5.4)
To come back to the household applications, the timed control kernel finds use in timing telephones or the television. Using the timed action module, the phone can be disabled at preprogrammed times. Through integration of the multicast module, several targets can be controlled by a timer, for example when the television display and its sound system are turned of using a timer.

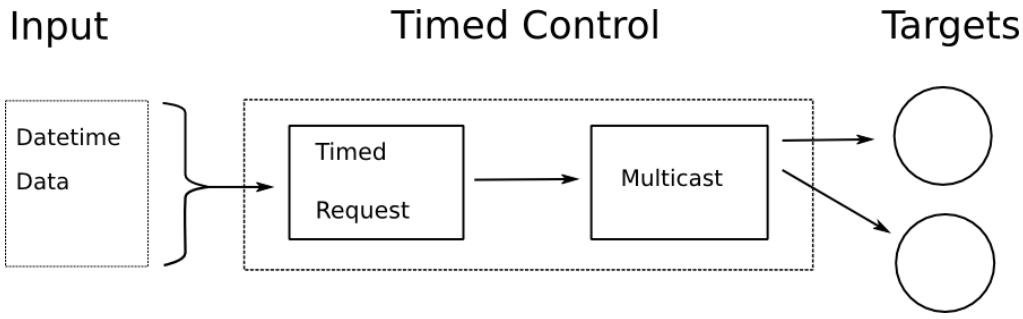

Figure 5.4: Application Kernel Timed Control.

5.1.5 Periodic Control

The periodic control application class defines applications controlling targets using a periodic action mechanism. The period and data are provided through an input interface. The input data is processed by the **periodic action** module, which repeatedly triggers a predefined action. Many targets can be addressed using the **multicast** module. (Figure 5.5)
Periodic irrigation systems to water the garden are an example of household applications for this application kernel. The periodic action module turns on the irrigation system repeatedly as specified by the user. In order to start many water sprinklers at the same time, the multicast module sends the start signal to a selected collection.

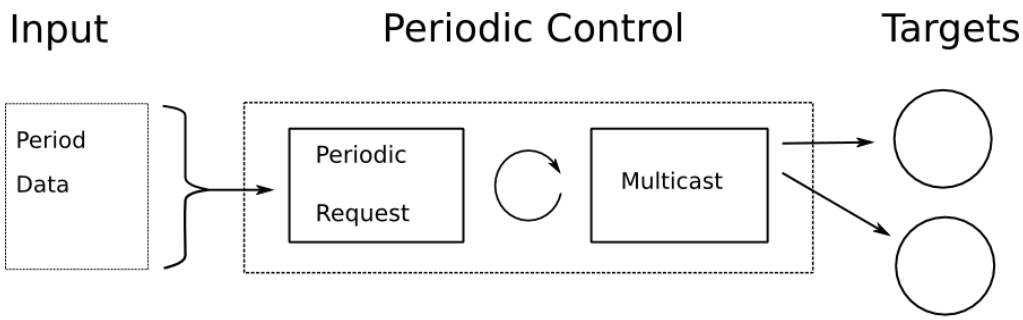

Figure 5.5: Application Kernel Periodic Control.

5.2 Detailed Case Study: Smart Thermostat

In order to evaluate the implemented modules and their assembly to application kernels, a smart thermostat application was implemented. To make the application more powerful yahoo's forecast is accessed through a web service.
Not every module will be incorporated into the smart thermostat. Nevertheless it gives an idea on how to use them and makes it possible to infer usability issues. The final application is not a single closed project, but a collection of apps and modules running on the Actinium App Server. New features to the smart thermostat can easily be added through additional apps responsible for new functionality.
The current implementation of the smart thermostat which is an assembly of three applications, incorporates three different application kernels. A first implementation using the **data collection and visualization** kernel gathers temperature information from the internet and thermostats. A second collection of apps and modules controls different temperatures for day and night using the

timed control mechanism. Ultimately, a third functionality prevents the indoor temperature to drop below a specified temperature by incorporating the **control system** application kernel.

5.2.1 Yahoo Forecast

The smart thermostat application uses the weather forecast to set temperatures. To access information about the weather forecast, an app connects to the yahoo forecast periodically and saves the information. All relevant information can then be accessed through a RESTful interface using CoAP requests. The app runs just like the modules in the Actinium App Server.

5.2.2 General Temperature Visualization

The **persisting service** is used to gather temperature information from the thermostat and the yahoo forecast web service. The temperature data can be retrieved from the database and is visualized in a timeline graph.

5.2.3 Day Night Control

The day night control mechanism can be used to set different temperatures for the day time and the night time. The exact start and end times of the day can be set by the user. To make it more interesting, daily sunrise and sunset time are being integrated. When the day start time was set before the sunrise, the temperature will be increased to compensate the missing sun warmth. The same happens in the evening, when sunset is before day end.
The new times to set the temperatures are evaluated every day at midnight. The **timed action module** is used to host the resulting requests until they are supposed to be executed. A day night control app creates a new timed action for each temperature to set. Sunrise and sunset are fetched from the yahoo forecast app before initializing the timed actions. When multiple thermostats are operated using this day night control mechanism, the **multicast** module targets all of them.

For demonstration purpose only, the temperature measured by the thermostat is stored in a database using the persisting service. In addition the persisting service also observes and stores the outdoor temperature and the times of sunrise and sunset.

5.2.4 Vacation Control

The third small application to prevent the temperature to drop below a minimum is implemented in the vacation control app. The user specifies a minimum and maximum temperature. To hold the temperature, an outdoor threshold temperature can be set to trigger an increase of heating indoors.
When the current outdoor temperature drops below the outdoor threshold, the maximum temperature is used to heat, otherwise the indoor temperature is kept at the minimum temperature. To make it more adaptive in case of a rapid outdoor temperature decrease, the forecast's minimum temperature is used. Whenever the forecast predicts a temperature below the outdoor threshold temperature, the indoor temperature is set to the middle temperature of the minimum and maximum. This way, whenever the forecast holds true, the time to heat to the maximum was reduced.
The **multiple aggregate** module collects both the current and the forecast temperature from the yahoo forecast app. It passes the unchanged values to the **control loop module** together with the source information. The control loop takes this information and decides on the new indoor temperature and sends it to the thermostat. Several thermostats can be controlled by sending the new indoor temperature to the **multicast** module, from where all the thermostats are targeted.

In order to preserve a history for demonstration purpose only, the temperature changes, indoor temperature, the current outdoor temperature, and the forecast are recorded using the persisting service.

5.2.5 Web Frontend

Through the web frontend (Section A.2) the user has full access to the functionalities of the smart thermostat. All three applications, the general temperature visualization, the day night control mechanism, and the vacation control mechanism, can be started and controlled using the frontend. With a simple mouse click, the persisting service can be run to gather information. At the same time, the temperatures and data stored in the database are being visualized in a annotated timeline chart.

Chapter 6

Conclusion

Through detailed study of existing applications for the Internet of Things, amongst others on wireless sensor networks, we found a classification into application classes. For the classification, we built a requirements hierarchy, as a tool to find similarities amongst the applications. Finally each application was assigned to exactly one application class.
The modules are implemented in Java or JavaScript using the Californium framework, a CoAP implementation, and Actinium as App Server. Data is stored sing CouchDB, a specifically suitable database architecture for distributed systems, such as the Internet of Things.
We were able to find a suitable collection of reusable modules to realize the functionality of each application class. A detailed case study on smart thermostats shows successful use of the modules and their assembly into application kernels.

Future work should focus on more applications for the smart household appliances. More extensive use of the modules, and hence the application kernels, over a longer period of time could bring up new issues. When necessary, the set of modules can easily be extended through new apps for Actinium or stand-alone applications using the Californium framework.

Bibliography

[1] Kovatsch, M.: Demo abstract: Human-coap interaction with copper. In: Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2011), Barcelona, Spain. (June 2011)

[2] Shelby, Z., Hartke, K., Bormann, C., Frank, B.: Constrained application protocol (coap). In: draft-ietf-core-coap-08. (2011)

[3] Fielding, R.T., Taylor, R.N. In: Modern web architecture. Volume 2. ACM, New York, NY, USA (May 2002) pages 115–150

[4] Kovatsch, M., Weiss, M., Guinard, D.: Embedding internet technology for home automation. In: Prodeedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010), Bilbao, Spain. (September 2010)

[5] Kovatsch, M.: Firm firmware and apps for the internet of things. In: Prodeedings of the 2nd ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA 2011), Honolulu, HI, USA., New York, NY, USA, ACM (May 2002) pages 61–62

[6] Pauli, D., Obersteg, D.I.: Californium. Master's thesis, ETH Zürich, Institute for Pervasive Computing, Department of Computer Science (2011)

[7] Lanter, M.: Actinium: An app server for the smart home. Master's thesis, ETH Zürich, Institute for Pervasive Computing, Department of Computer Science (2012)

[8] Lanter, M., Kovatsch, M.: Coaprequest api. Master's thesis, ETH Zürich, Institute for Pervasive Computing, Department of Computer Science (2012)

[9] Mottola, L., Picco, G.P. In: Programming wireless sensor networks: Fundamental concepts and state of the art. Volume 43 number 3. ACM (April 2011) pages 1–51

[10] Szewczyk, R., Osterweil, E., Polastre, J., Hamilton, M., Mainwaring, A., Estrin, D. In: Habitat monitoring with sensor networks. Volume 47, number 6. ACM (June 2004)

[11] Maroti, M., Simon, G., Ledeczi, A., Sztipanovits, J. In: Shooter Localization in Urban Terrain. Volume 37, issue 8. IEEE, Dept. of Electr. Eng. and Comput. Sci., Vanderbilt Univ., Nashville, TN, USA (August 2004) pages 60–61

[12] Thorstensen, B., Syversen, T., Bjørnvold, T.A., Walseth, T.: Electronic shepherd: a low-cost, low-bandwidth, wireless network system. In: Proceedings of the 2nd international conference on Mobile systems, applications, and services, Boston, MA, USA. (June 2004)

[13] Kwong, K.H., Wu, T.T., Goh, H.G., Stephen, B., Gilroy, M., Michie, C., Andonovic, I. In: Wireless Sensor Networks in Agriculture: Cattle Monitoring for Farming Industries. Volume 5, number 1. Piers Online, Centre for Intelligent Dynamic Communications, Department of Electronic and Electrical Engineering, University of Strathclyde, Glasgow, UK (2009) 31–35

[14] Sheth, A., Thekkath, C.A., Mehta, P., Tejaswi, K., Parekh, C., Singh, T.N., Desai, U.B. In: Senslide: a distributed landslide prediction system. Volume 41, number 2. ACM (April 2007) pages 75–87

[15] Wittenburg, G., Terfloth, K., Villafuerte, F.L., Naumowicz, T., Ritter, H., Schiller, J.: Fence monitoring: experimental evaluation of a use case for wireless sensor networks. In: Proceedings of the 4th European conference on Wireless sensor networks, Delft, The Netherlands. (January 2007)

[16] Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., Welsh, M.: Fidelity and yield in a volcano monitoring sensor network. In: Proceedings of the 7th symposium on Operating systems design and implementation, Seattle, Washington. (November 2006)

[17] Mottola, L., Picco, G.P., Ceriotti, M., Guna, S., Murphy, A.L. In: Not all wireless sensor networks are created equal: A comparative study on tunnels. Volume 7, number 2. ACM (August 2010) pages 1–33

[18] Hoogenboom, G.: Weather monitoring for management of water resources. In of Ecology, I., ed.: Proceedings of the 2001 Georgia Water Resources Conference, Universitiy of Georgia, Athens, Georgia, Department of Biological and Agricultural Engineering, Universitiy of Georgia, Kathryn J. Hatcher (March 2001)

[19] Serby, V.M.: Timer Control for Telephone, Woodmere, N.Y. (may 1990)

[20] Maclay, W.R., Hewett, W.A.: Timer Control for Television, Pentalux Corporation, Mountain View, California. (May 1986)

[21] Bittanti, S., Fronza, G., Guardabassi, G. In: Periodic Control: A Frequency Domain Approach. Volume AC-18, number 1. IEEE (January 1973) pages 33–38

[22] Jiang, X., Dawson-Haggerty, S., Dutta, P., Culler, D.: Design and implementation of a high-fidelity ac metering network. In: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks. (April 2009) pages 253–264

[23] Polizzi, D.D.: system for automatic periodic irrigation, Upland, California. (March 1974)

[24] Milenković, A., Otto, C., Jovanov, E. In: Wireless sensor networks for personal health monitoring: Issues and an implementation. Volume 29 number 13-14. ACM (August 2006) pages 2521–2533

Appendix A

Appendix Chapter

A.1 CouchDB

A.1.1 View Representation

```
function(doc) {
   if (doc.device && doc.value!=null && is_numeric(doc.value)) {
      emit(doc.device, doc.value);
   }

   function is_numeric(input){
         return typeof(input)=='number';
   }
}
```

Figure A.1: Map function to create view index.

```
function(keys, values, rereduce) {
   if (rereduce) {
      var result = {"device":values[0].device,
                "sum":values[0].sum};
      for(var i=1,e=values.length; i<e; ++i) {
                        result.sum = result.sum + values[i].sum;
                };
      return result;
   } else {
      var result = {"device": keys[0][0],
                "sum":values[0]};
      for(var i=1,e=keys.length; i<e; ++i) {
                        result.sum = result.sum + values[i];
                };
      return result;
   }
}
```

Figure A.2: Reduce function with rereduce to create view index.

A.2 Web Frontend Screenshots

Screenshots from the web fronted.

A.2.1 Smart Thermostat

- Input - Figure A.3
- History - Figure A.4
- Graph - Figure A.5
- Multiple Aggregate Task - Figure A.6
- Control Loop Task - Figure A.7

A.2.2 Persisting Service

- Create Instance - Figure A.8
- Instance - Figure A.9
- Data Retrieval - Figure A.10

A.2.3 Timed Action

- Create Task - Figure A.11
- Tasks -Figure A.12

A.2.4 Periodic Action

- Create Task - Figure A.13
- Tasks - Figure A.14

A.2.5 Multicast

- Create Task - Figure A.15
- Tasks -Figure A.16

A.2.6 Multiple Aggregate

- Create Task - Figure A.17
- Tasks -Figure A.18

A.2.7 Control Loop

- Create Task - Figure A.19
- Tasks -Figure A.20

A.2.8 Push Simulation

- Create Task - Figure A.21
- Tasks -Figure A.22

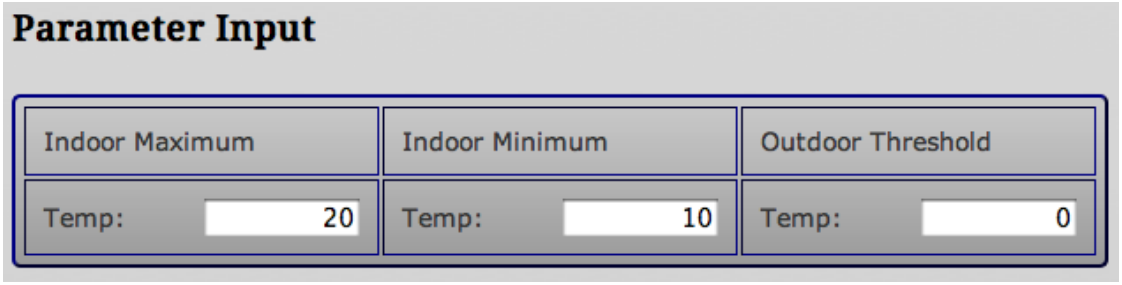

Figure A.3: Smart Thermostat - Input (here Vacation Control)

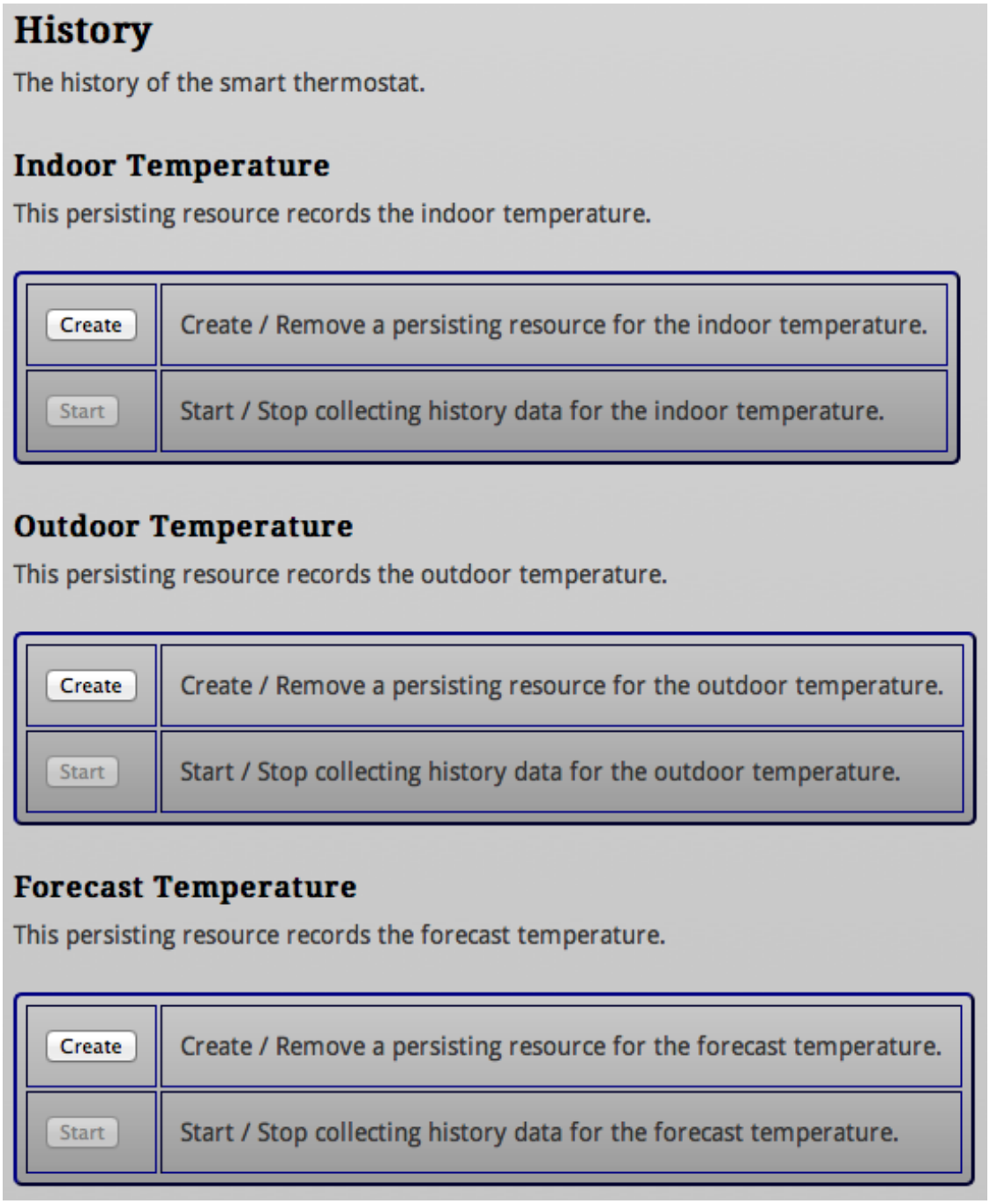

Figure A.4: Smart Thermostat - History

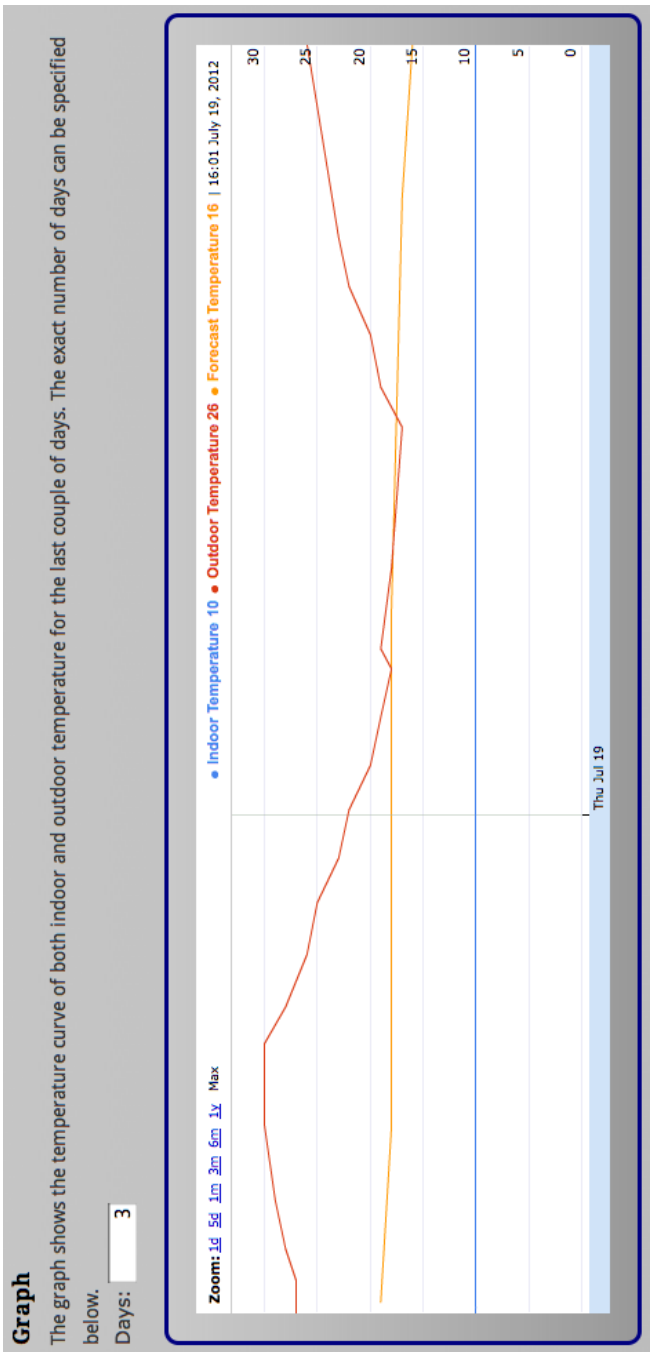

Figure A.5: Smart Thermostat - Graph (here Vacation Control)

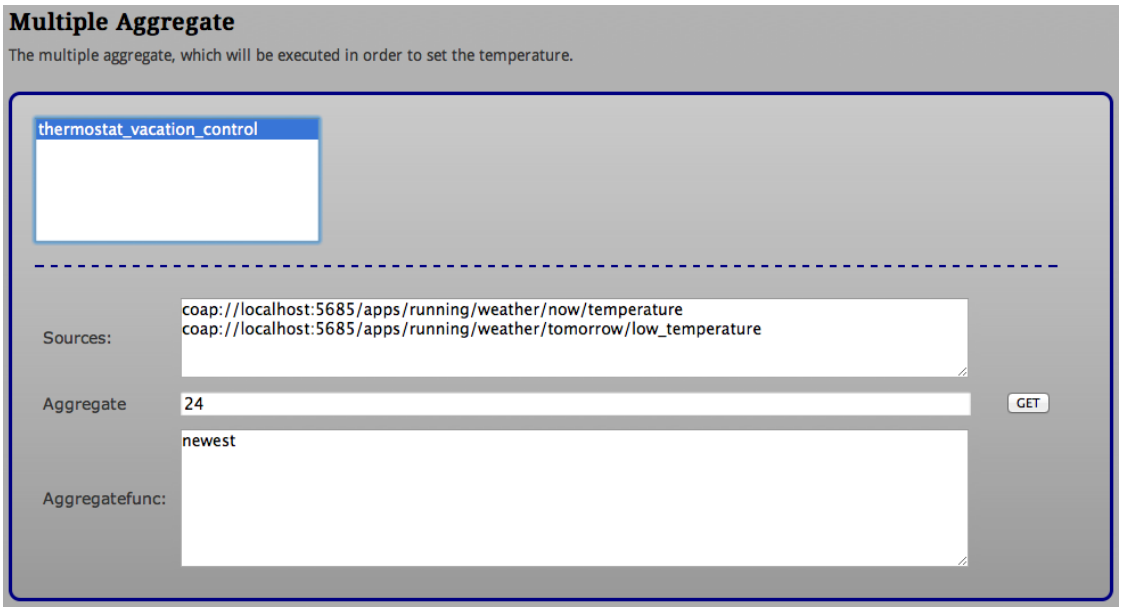

Figure A.6: Smart Thermostat - Multiple Aggregate Task (here Vacation Control)

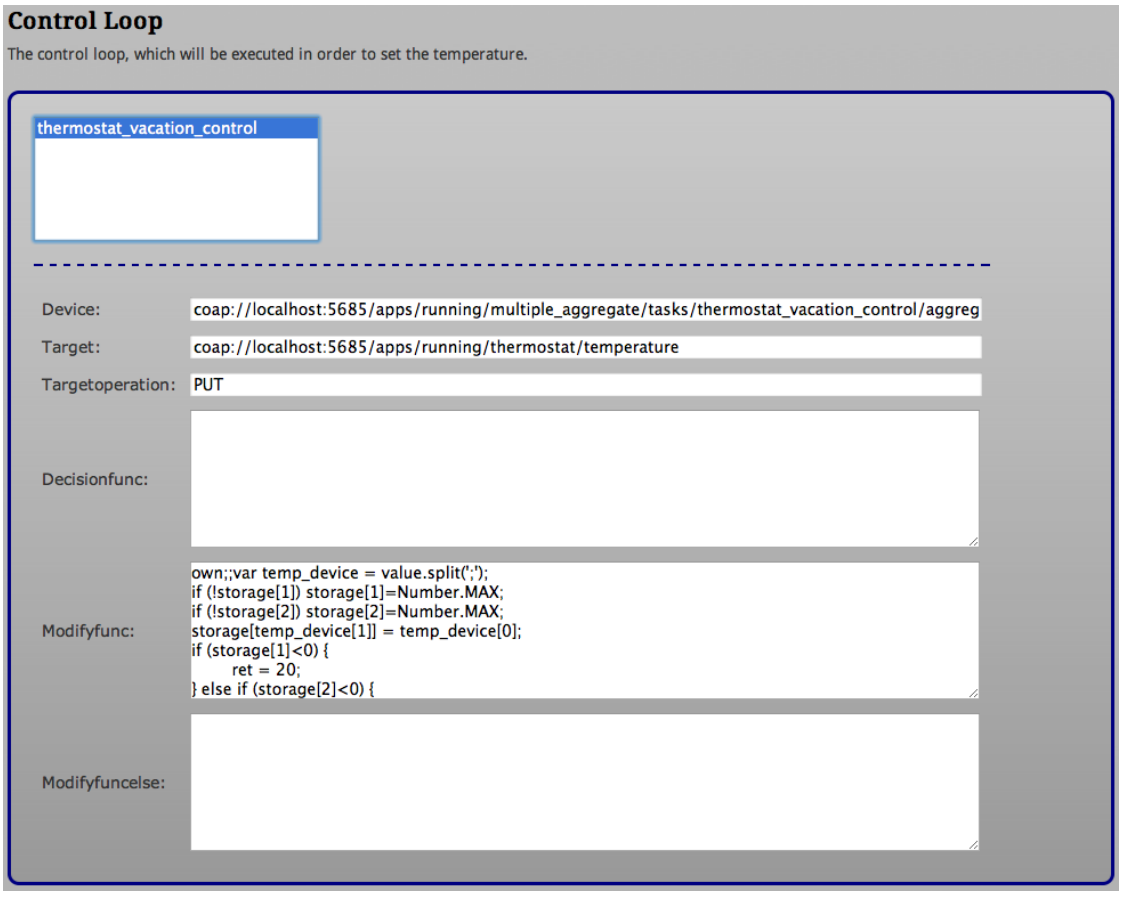

Figure A.7: Smart Thermostat - Control Loop (here Vacation Control)

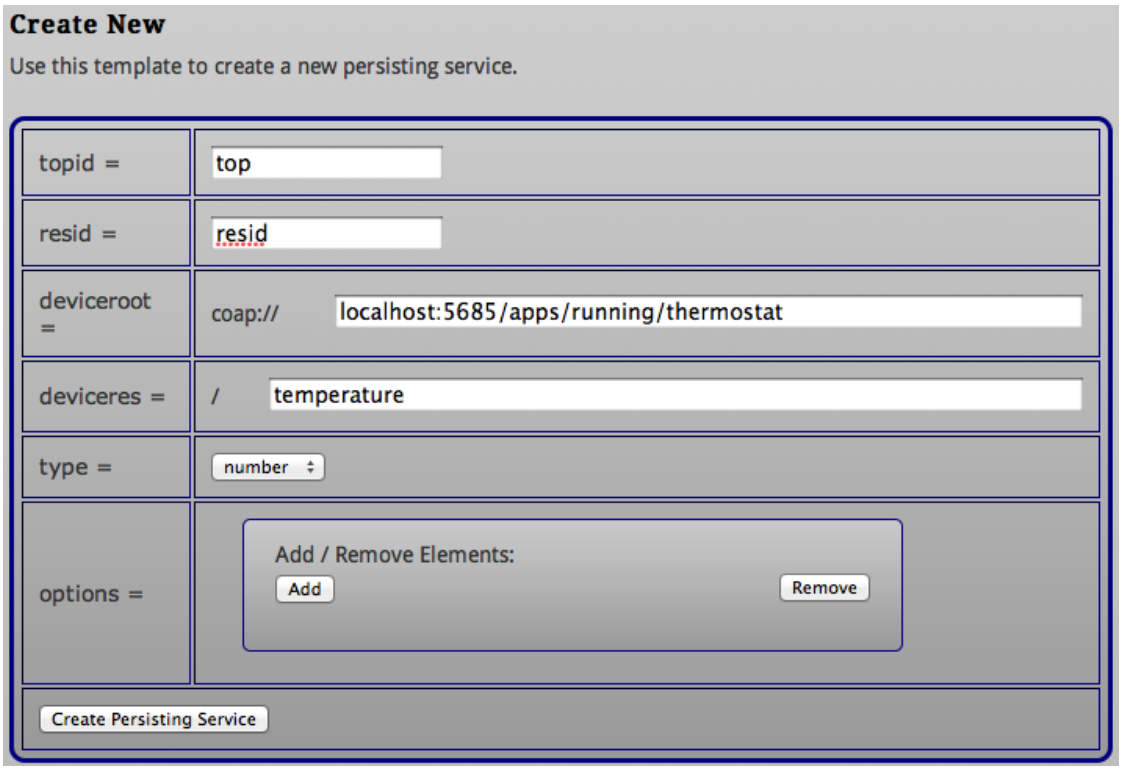

Figure A.8: Persisting Service - Create

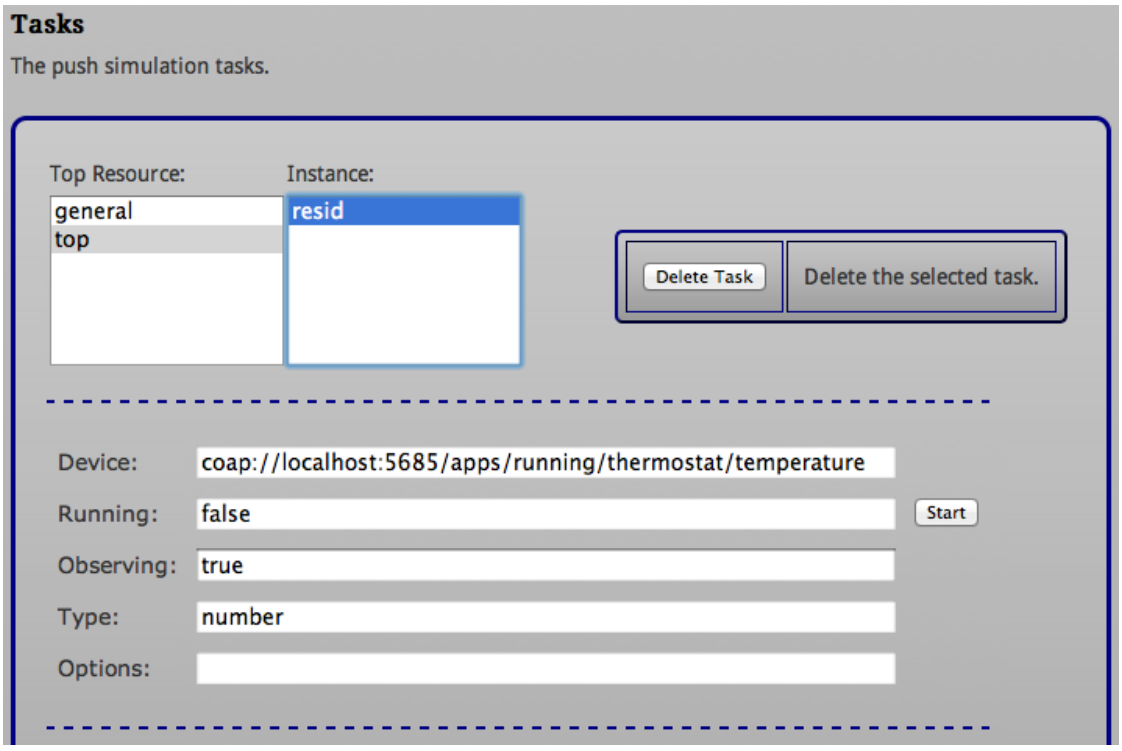

Figure A.9: Persisting Service - Task

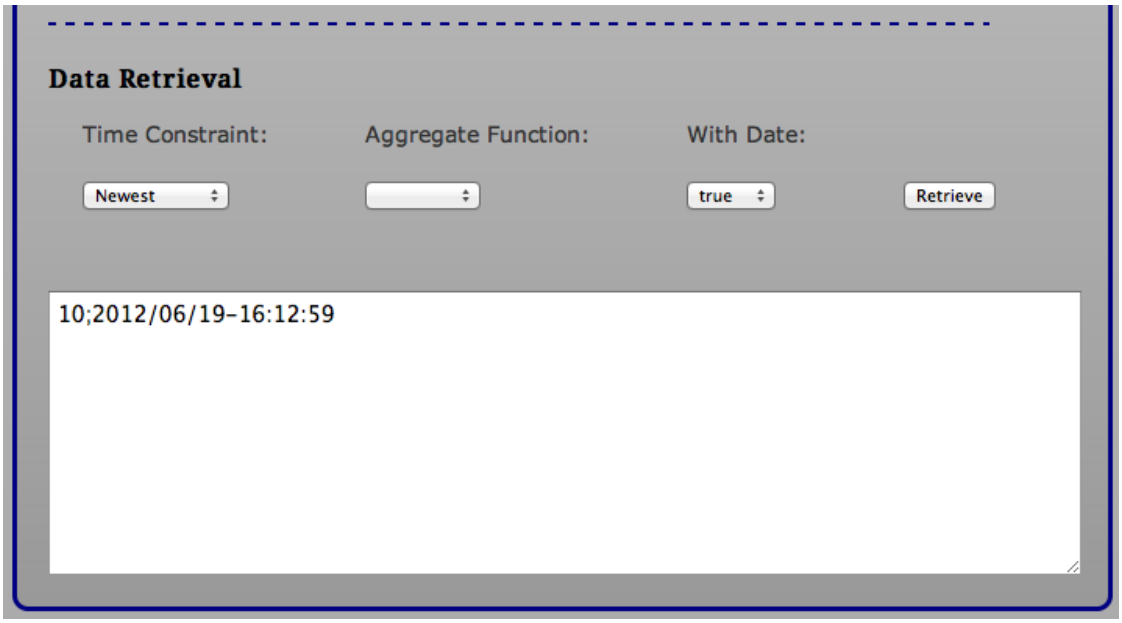

Figure A.10: Persisting Service - Data Retrieval

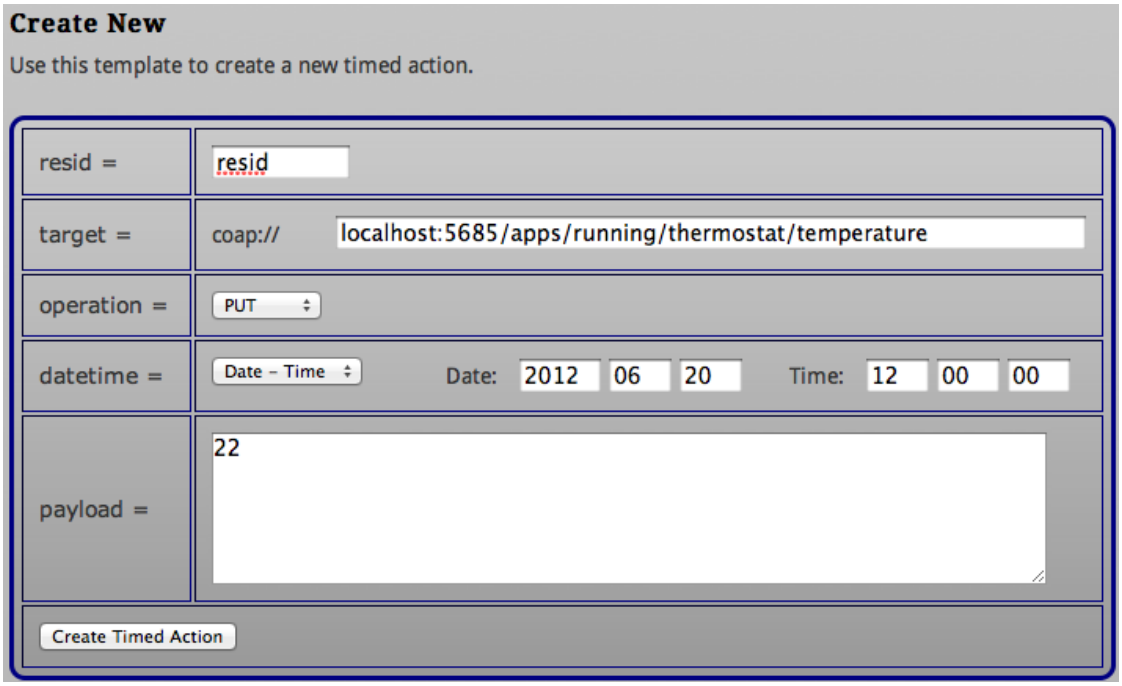

Figure A.11: Timed Action - Create

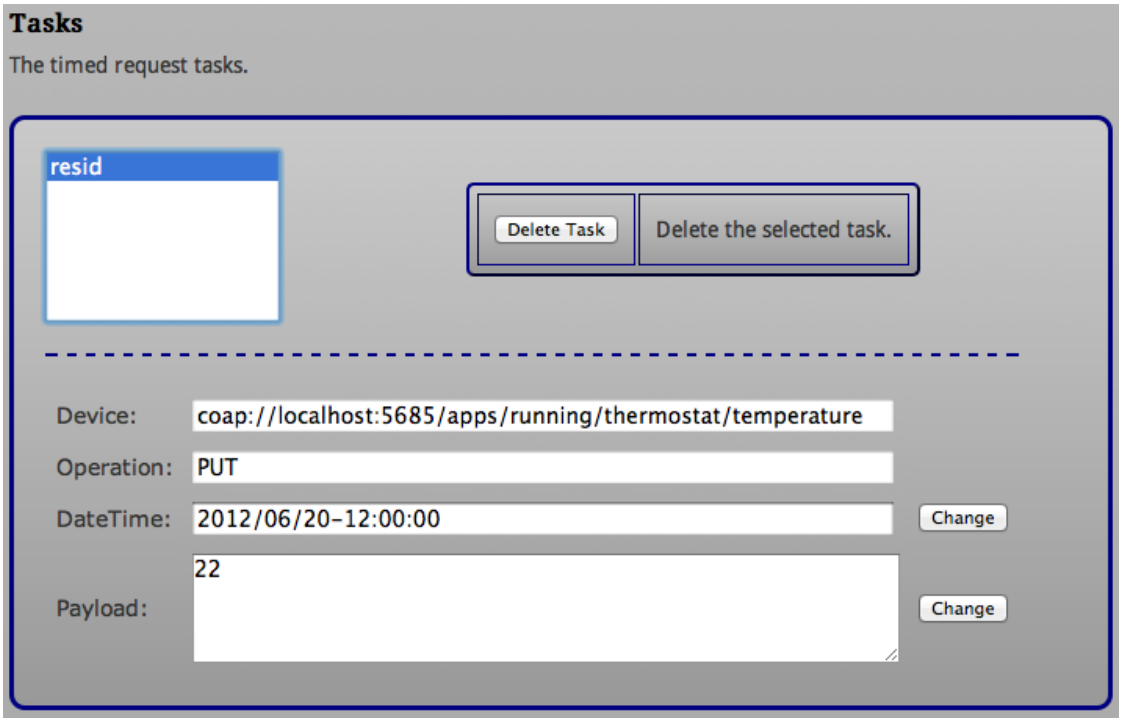

Figure A.12: Timed Action - Task

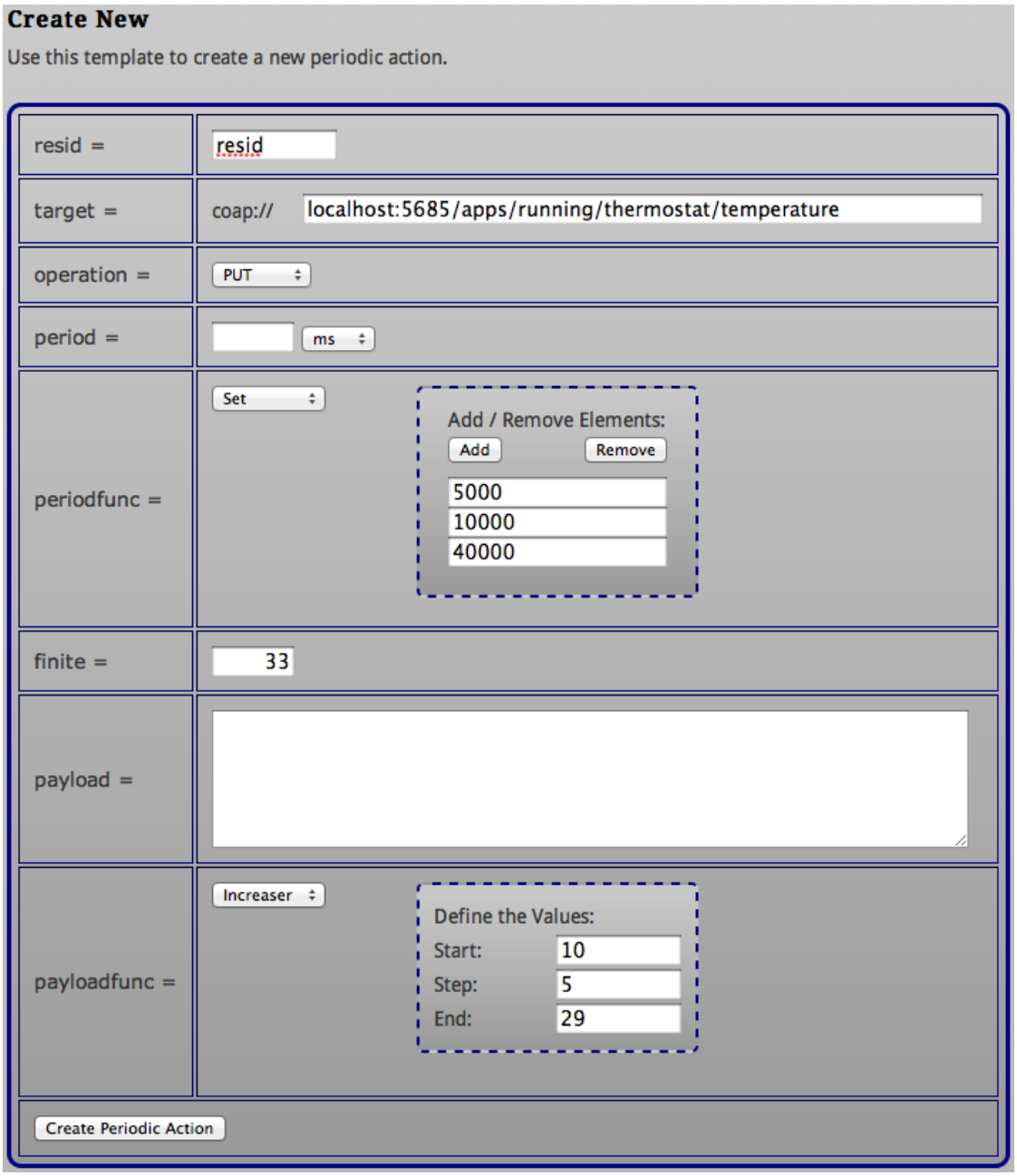

Figure A.13: Periodic Action - Create

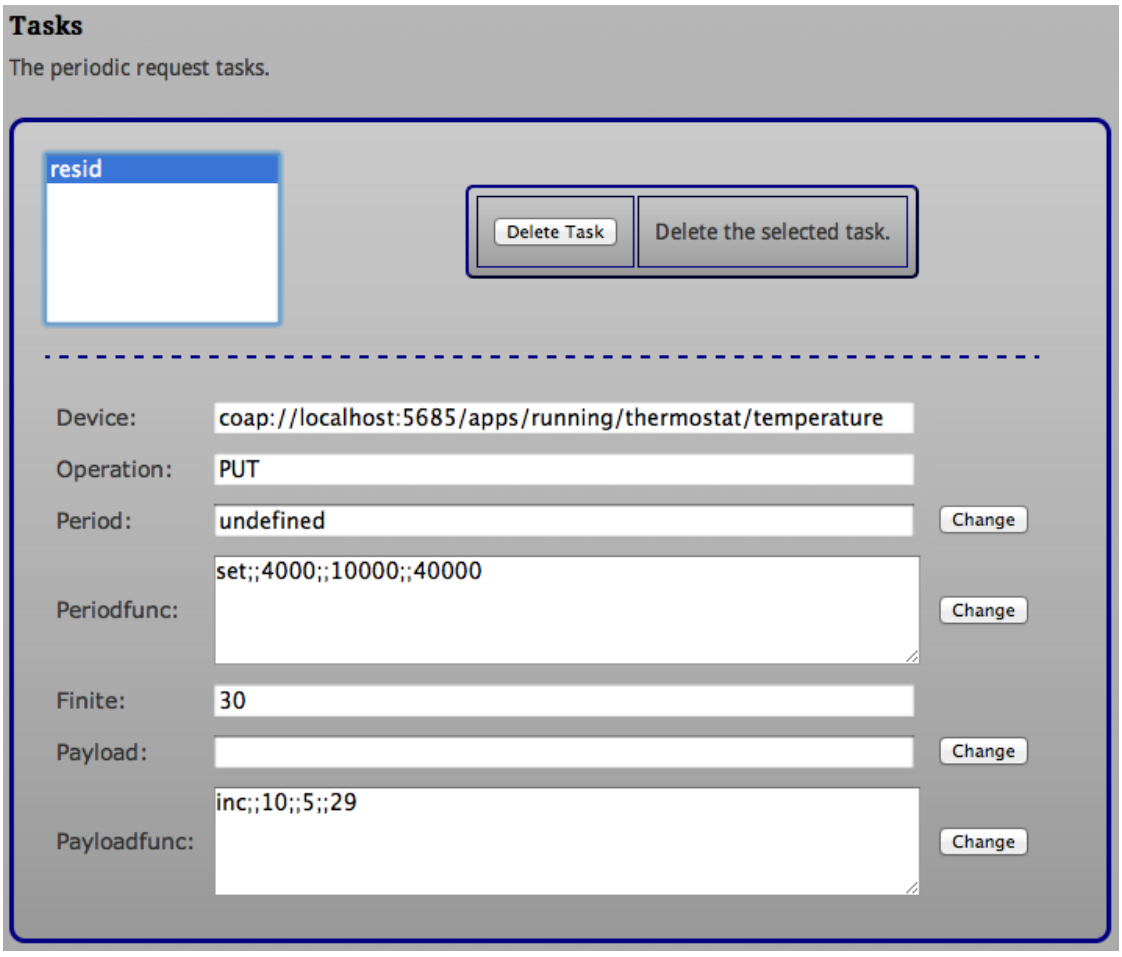

Figure A.14: Periodic Action - Task

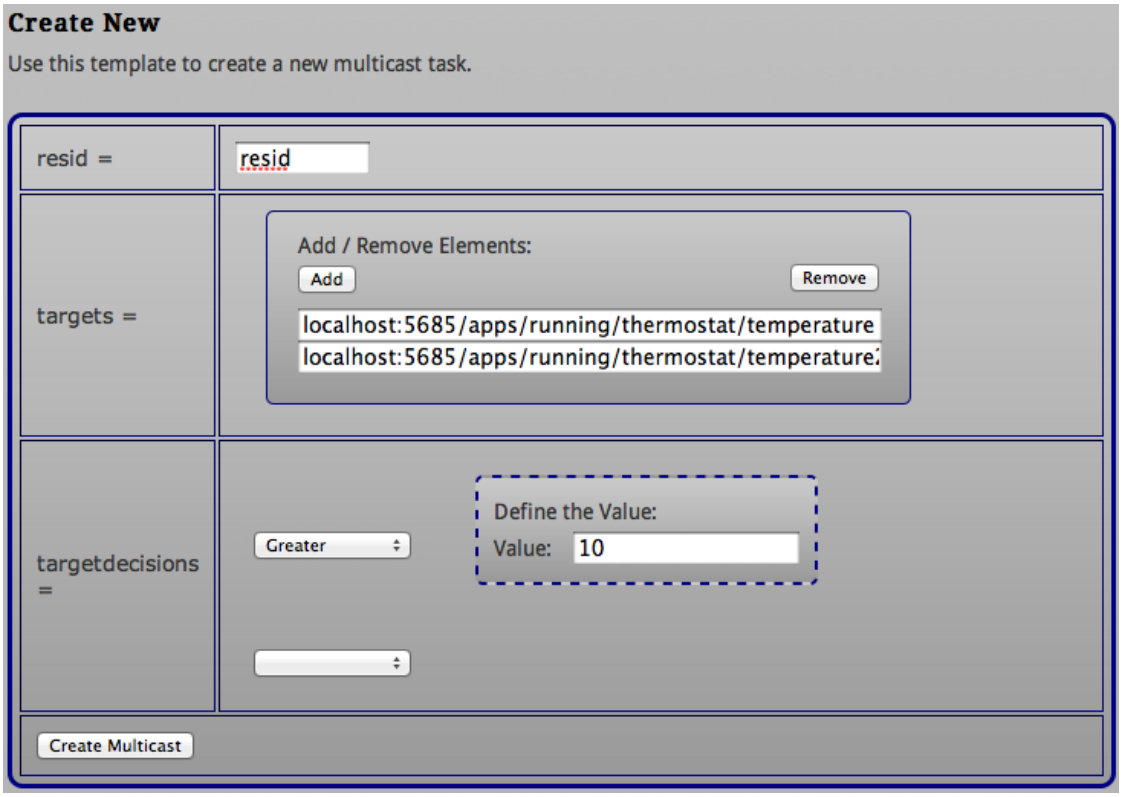

Figure A.15: Multicast - Create

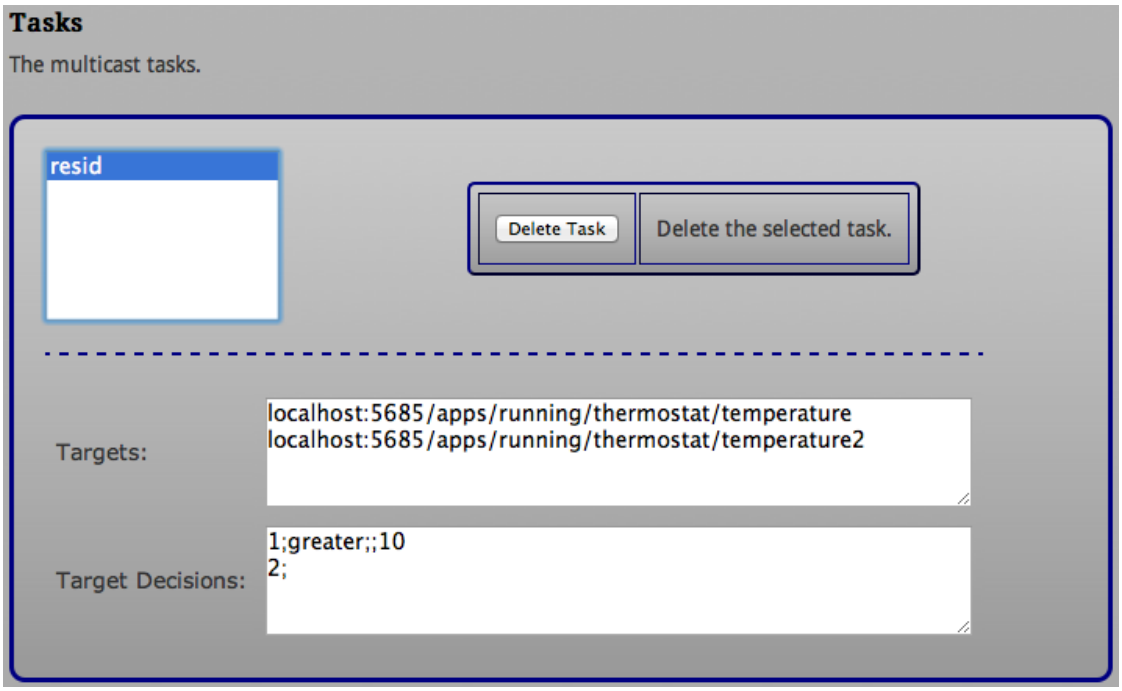

Figure A.16: Multicast - Task

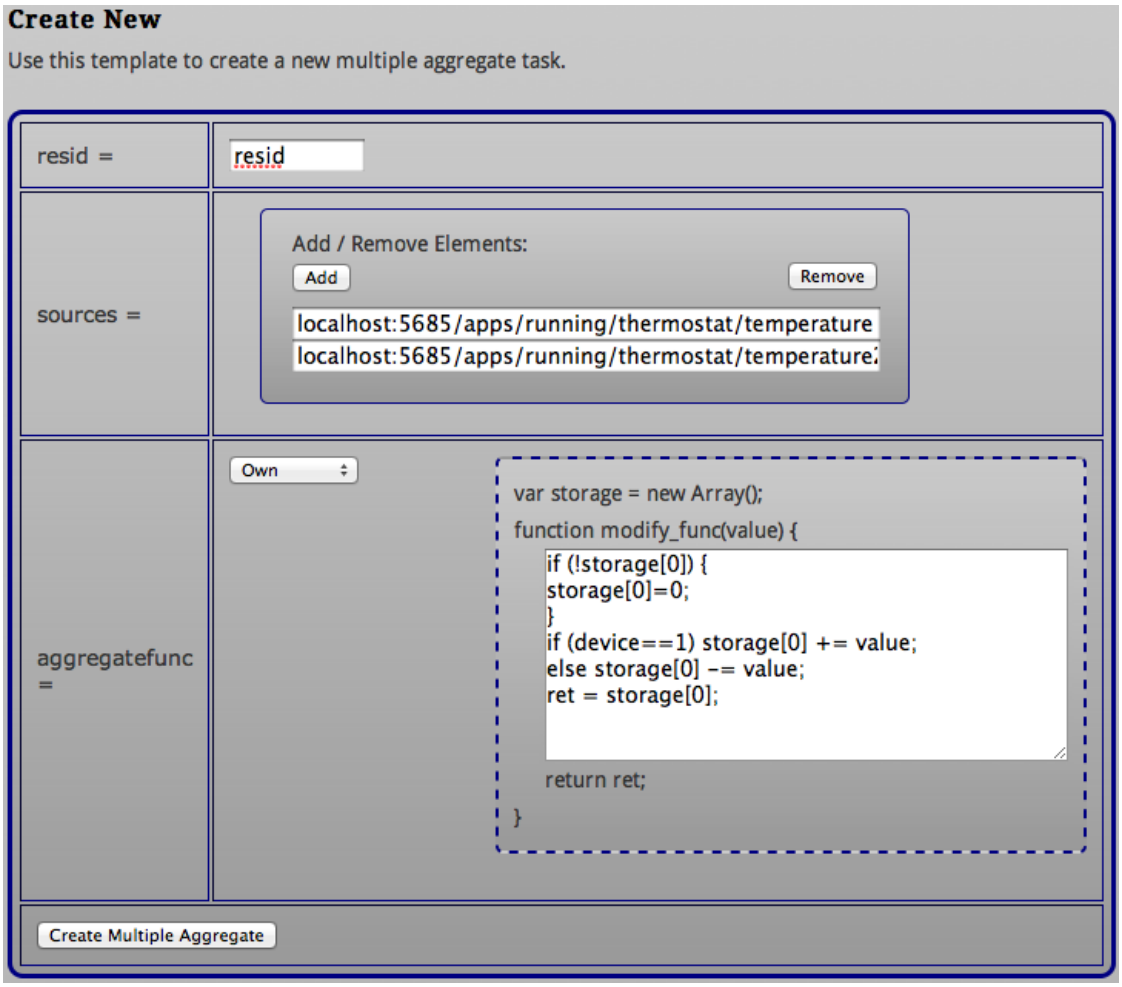

Figure A.17: Multiple Aggregate - Create

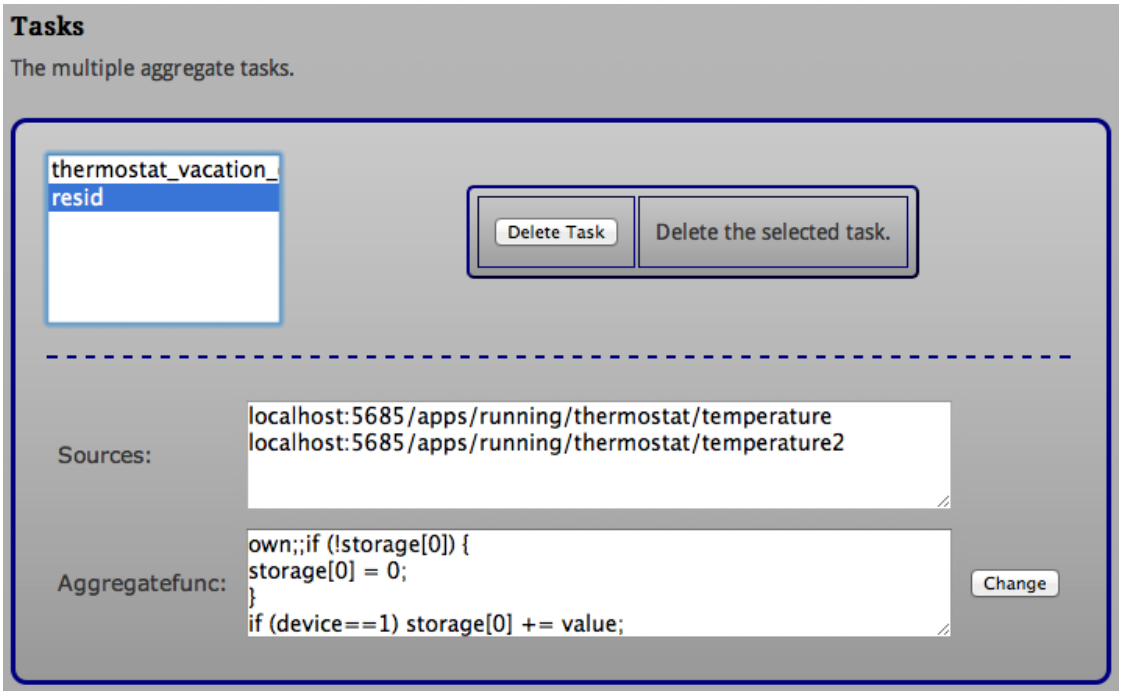

Figure A.18: Multiple Aggregate - Task

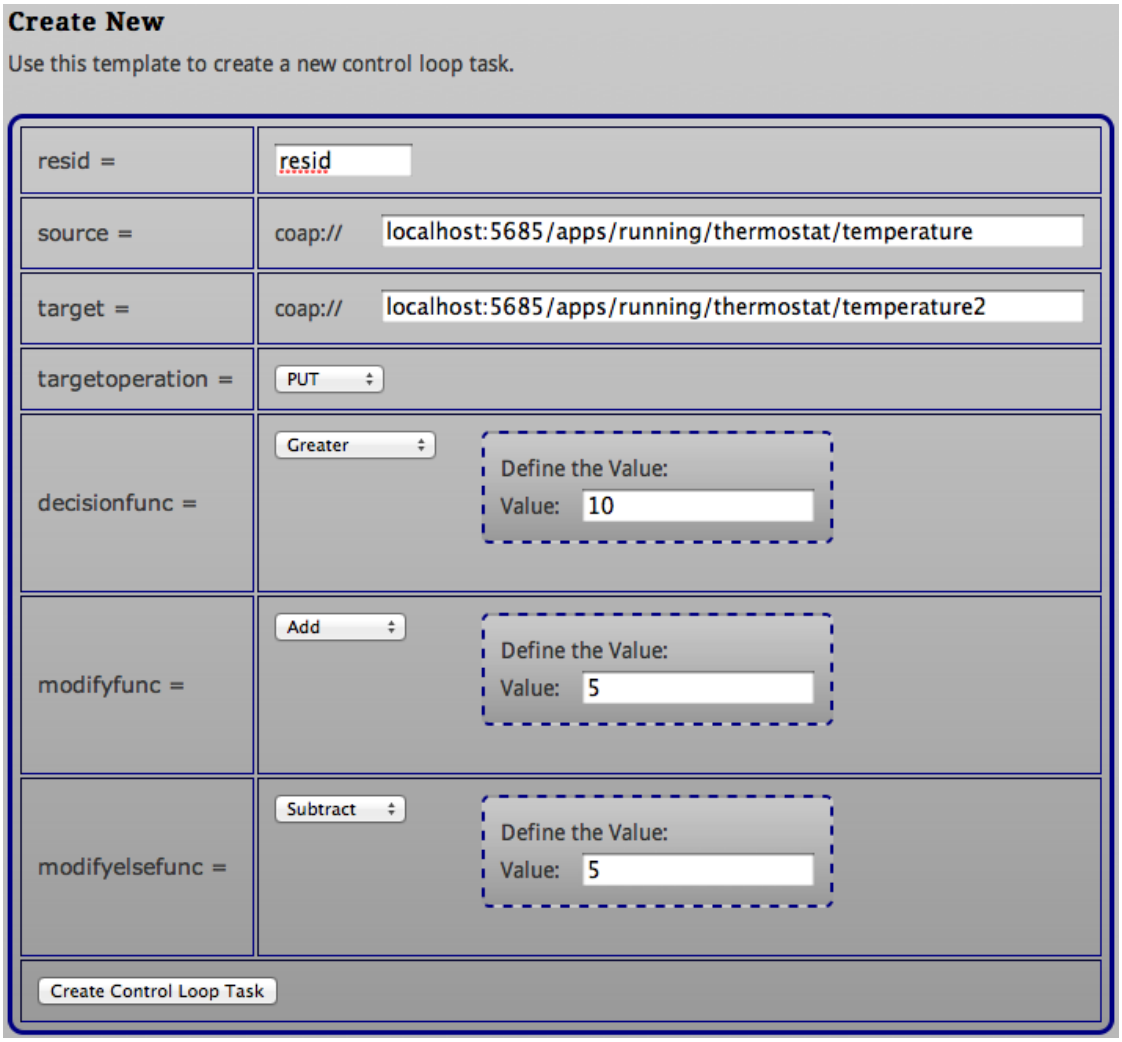

Figure A.19: Control Loop - Create

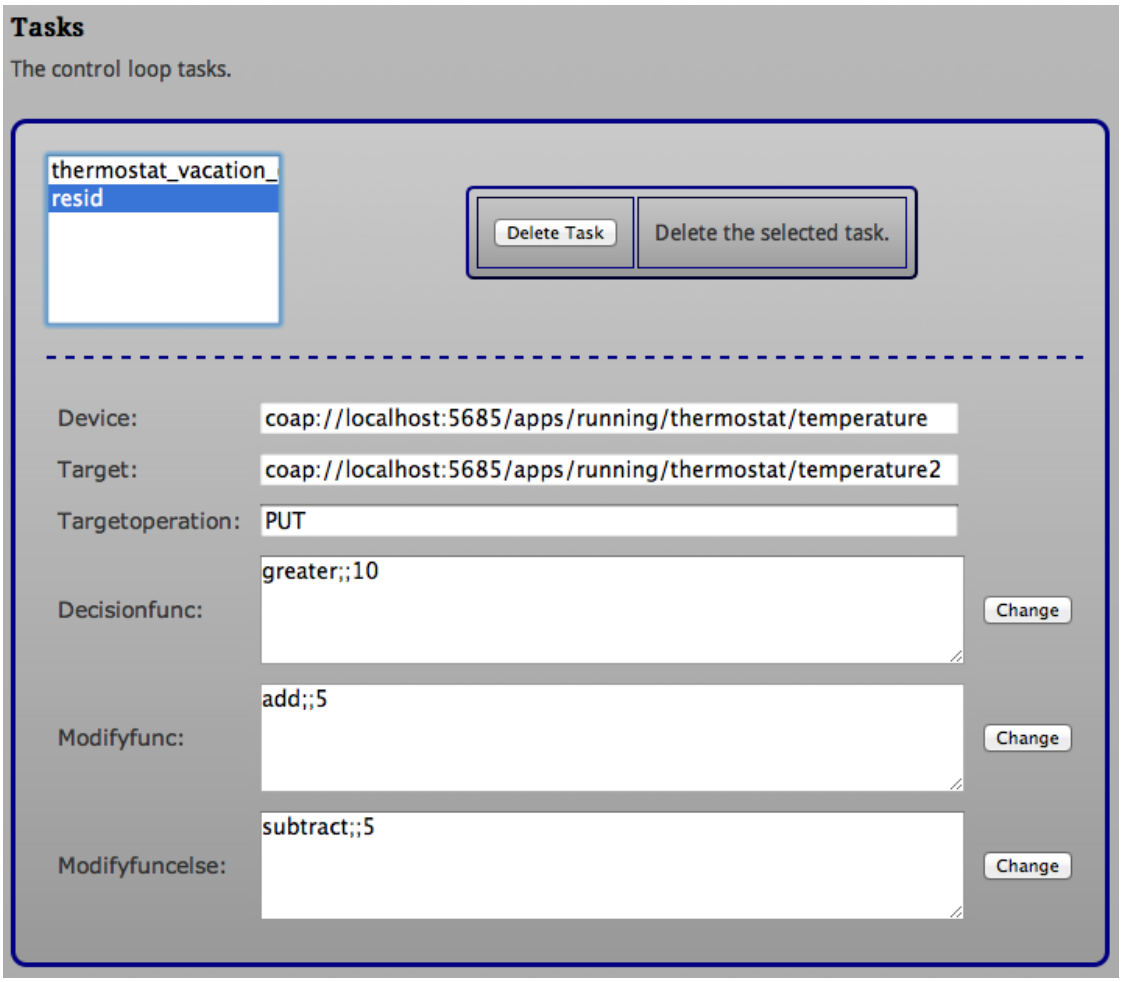

Figure A.20: Control Loop - Task

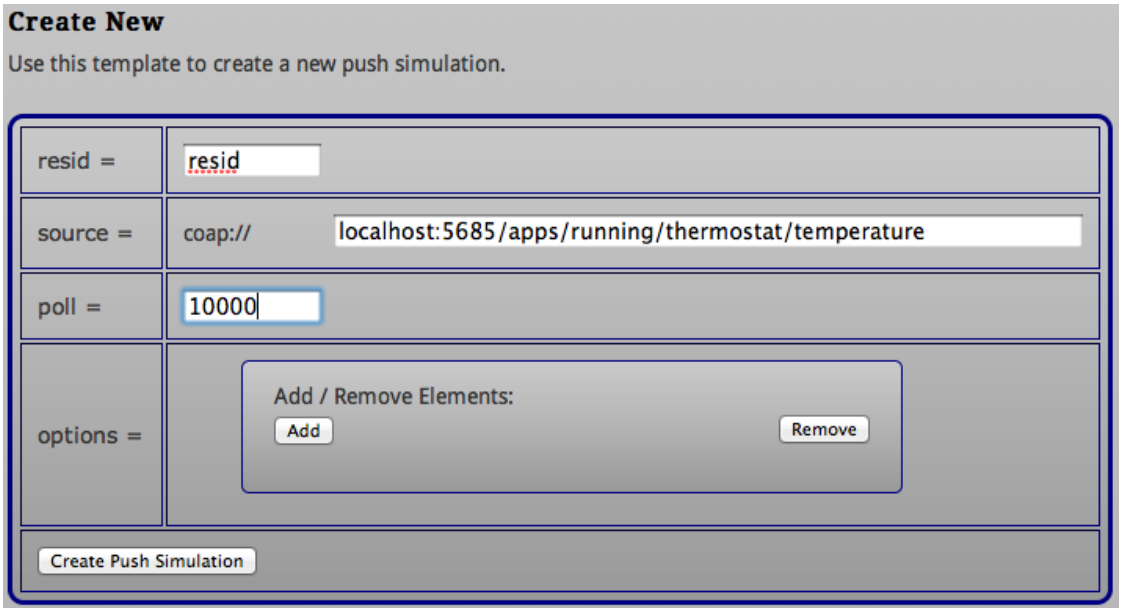

Figure A.21: Push Simulation - Create

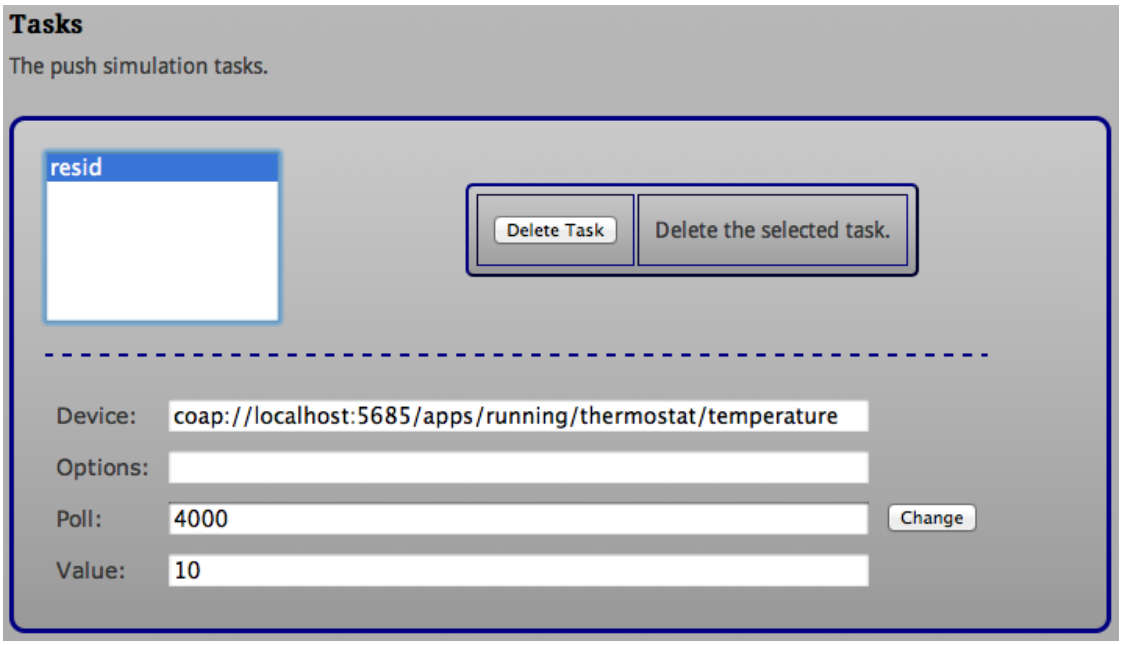

Figure A.22: Push Simulation - Task