FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Application Programming Interface for Constrained RESTful Environments based on a Groovy Domain Specific Language

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Computer Science

eingereicht von

Peter Klein
Matrikelnummer 8251105

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dr. Wolfgang Kastner
Mitwirkung: DI Markus Jung

Wien, 30.04.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Application Programming Interface for Constrained RESTful Environments based on a Groovy Domain Specific Language

BACHELOR WORK

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Science

by

Peter Klein
Registration Number 8251105

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ. Prof. Dr. Wolfgang Kastner
Assistance: DI Markus Jung

Vienna, 30.04.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Peter Klein
Kuhngasse 8, Haus 5, 2201 Gerasdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Abstract

Introduction

Aim of this thesis it to design an develop an API for constrained RESTful environments based on a domain-specific language. The API provides access to the data and operations of IoT (Internet of the Things) devices communicating directly via IPv6 or through a multi-protocol gateway that connects to existing building automation systems based on standards like KNX or BACnet.

In the proposed IPv6 based IoT architecture each device respectively each gateway is assigned an own static IP address and runs a server that exposes a set of IoT objects to other IoT devices and to applications. The IoT objects are described according to the oBIX (Open Building Information Xchange) standard as a set of oBIX objects based on oBIX contracts. An oBIX contract provides a template for a certain class of objects, e.g. the class of all light switches. HTTP and CoAP (Constrained Application Protocol) bindings allow any application to access the IoT devices by one of these protocols. Application design based on direct access via HTTP or CoAP however requires knowledge of these protocols and implementation of a corresponding protocol client in every application. The API described in the thesis provides access to the devices by a client application simply by accessing a local proxy object working as a device abstraction. All communication protocol specific functionality and support of encoding is encapsulated in the API. In addition to getting and setting an object's values, oBIX standard functionality for watches, lobby, history and history rollup is provided together with support for CoAP observe functionality. The domain specific classes are generated from oBIX contracts similar to wsdl2java for web services so that arbitrary sets of contracts are supported. The thesis describes the concepts, implementation and usage of the API.

Contents

CHAPTER 1

Introduction

1.1 Internet of the Things

The basic idea of the Internet of Things (IoT) is to give entities in the physical world a corresponding representation in the information world so that each of these entities exposes information about its type, structure, data and functions it can perform. Compared with the current Internet where most of the information is provided by humans and used by humans in IoT [20] most of the information is provided and consumed by machines which poses a set of basic issues:

- *Large Number of Objects:* The number of entities (called IoT objects) in the Internet of Things will clearly outnumber the number of entities in the current internet, expected to grow into 50 billion entities by 2020. Each of these IoT objects has to be uniquely identifiable which will exceed the range of available IPv4 addresses. IPv6 [1] due to its extremely large address space would be able to cope with the demand. Alternative identification schemes are unique numbers such as used in RFID tags and the scheme used in the semantic web which intends to make all things (not only physical entities) addressable by a unique URI.

- *Interoperability:* In addition to being able to identify the IoT objects in the physical world it is also necessary to enable communication between them and ensure interoperability. IoT objects either communicate with each other in peer to peer manner or they communicate with central server based objects. They must be able to work together, even if their interoperability is not designed explicitly as in a single application, but is guaranteed by following a set of standards. IoT applications can use any objects adhering to these standards.

- *Semantics:* It has to be taken into account that machines do not have the human capabilities of interpretation, so precise foundations of semantics have to be assigned to the objects, their data and functionalities. Ontology frameworks such as SensorML [10] or DomoML [19] can provide such a foundation.

The term *Internet of Things* is not really precisely defined and overlaps with concepts such as *Ambient Intelligence* which is defined as systems that are sensitive and responsive to people, e.g. sensing in a building automation system the presence of a person in a room and setting the light and the room temperature accordingly, *Ubiquitous computing*, which means that computing devices are fully integrated into the physical space and people are using them without being aware of it, and *Machine-to-Machine* which focuses on the communication between devices equipped with wireless or wired communication capabilities and servers running specific applications such as fleet management or machine monitoring.

There are numerous applications envisioned in the IoT world in domains like Smart Cities, Smart Environment, Smart Grids, Logistics, Health or Agriculture. According to *6LowPAN: The Wireless Embedded Internet* [17] examples are:

- *Smart Roads:* Intelligent highways with warning messages and diversions according to climate conditions and unexpected events like accidents or traffic jams
- *Snow Level Monitoring:* Snow level measurement to know in real time the quality of ski tracks allows security corps to perform avalanche prevention
- *Smart Grid:* Measure and control energy consumption, production and distribution
- *Container Tracking:* Measurement of position, vibrations, forces, container openings or cold chain maintenance for insurance purposes
- *Patient Monitoring:* Measurement of vital functions such as heart rate, blood pressure or movement of critical patients at home
- *Green Houses:* Control of climate conditions to maximize the production of fruits and vegetables and their quality

1.2 Building Automation

Architecture

The term building automation comprises automation systems to monitor, control and optimize technical processes in a building such as heating, ventilation, air condition, energy and water supply or surveillance. Elements in such a process are sensors (measuring some physical quantity), actuators (to change behaviour of the process), user interfaces, consumers and other technical devices.

The architecture of a building automation system follows a layered approach in the following summary viewed bottom-up:

- *Field Level:* It consists of actuators (e.g. analogue values of temperature or pressure, digital values of contacts), sensors (valves and switches) and their cabling to the corresponding control system.

- *Control Level:* Individual digital control systems directly connected to sensors and actuators responsible for a specific task like control of room temperature. Communication between the control systems is either done manufacturer specific or by standardized bus systems such as LONWorks or KNX.
- *Management Level:* Single or distributed system performing visualization, supervision and optimization according to the term SCADA (Supervision, Control And Data Acquisition). Systems at the management communicate with the control level via standardized protocols such as BACnet [7] or OPC [9]. Management systems can also be distributed between local systems that perform local management autonomously and remote systems that communicate with the local management systems via IP connection.

A standard model for building automation systems is described in [2]. The following figure taken from [13] shows a typical example of a building automation system. The system consists of two independent field level secondary bus systems, each of them handling a set of control elements such as a boiler controller or a lighting controller. A PLC (programmable logic control) controls each of the bus systems and connects to a primary, management level bus system that connects to systems such as web servers and graphical workstations for users interacting with the system as shown in Fig. 1.1.

Technologies

A strong current focus is on interoperability so that field and control components of different manufacturers are working together based on common bus systems and protocols [15]. Each of the presented technologies is applicable to one or more of the levels described in the building automations model.

Technology	Field Level	Control Level	Management Level
KNX	X	X	-
LONWorks	X	X	-
BACnet	-	X	X
OPC	-	-	X
OSGI	-	-	X

Table 1.1: Technologies in Building Automation Levels

- *KNX (Konnex Bus)* [8] describes how sensors and actuators are connected via a bus system and communicate via a standardized protocol. It is the successor and compatible to EIB (European Installation Bus). KNX separates the control functions from the energy distribution on hardware level and can work with several communication systems such as twisted pair, power line, radio or Ethernet.
- *BACnet* was developed by the American Society of Heating, Refrigerating and Air Conditioning Engineers and is standardized by ISO as ISO 16484-5. BACnet defines a set

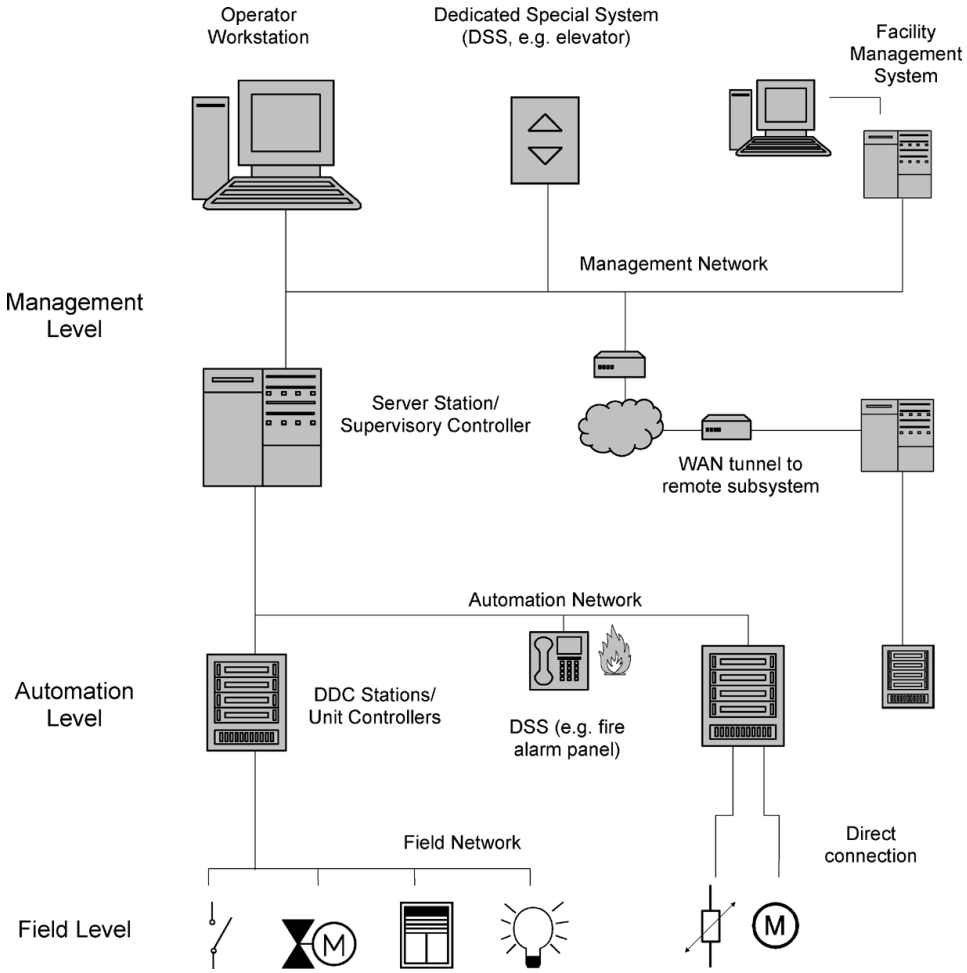

Figure 1.1: Building Automation System

of services such as data sharing, event and alarm processing, processing of value changes and device and network management. The standard defines a set of object types: analogue and digital input, analogue and digital output, analogue and digital values, notifications, trends, calendars and schedules. From communication protocols it follows a 4 layer model (Link Layer, MAC Layer, Network Layer and Application Layer) with several options for the layers 1 and 2 like Ethernet, BACnet/IP, point to point over RS232 or ZigBee for wireless communication.

- *LONWorks (Local Operation Network)* is a networking platform built on a protocol created by Echelon Corporation for networking devices over media such as twisted pair, powerlines, fiber optics, and RF. It was submitted to ANSI and accepted as a standard for control networking (ANSI/CEA-709.1-B). The underlying LONTalk protocol covers the layers 2-7 of the ISO reference model. LONWorks is used in building and home automa-

tion, transportation, utility, and industrial automation. It is a field level bus system, where devices called *Nodes* communicate over the bus. Nodes are categorized into sensors, actuators and controllers. From a logical point of view nodes communicate via communication objects called *Network Variables*. In order to facilitate interworking between nodes of different manufacturers standard network variable types are defined. Interfaces connect LONWorks devices to management level systems, e.g. via OPC.

- *OPC (OLE for Process Control)* is a standard for supplier independent communication in automation systems. It was created by a set of leading companies in the automation industry and is currently maintained by OPC foundation having more than 400 members. Basic idea is that each device provides an OPC driver which is then used by an OPC server to communicate with the devices. OPC clients such as a visualization system communicate with the OPC server and not with the individual devices. The standard comprises a set of services: Real-time Data Access, Alarms and Events, Historical Data Access, Data Exchange, Command Handling and Web Services. Devices and their functions are represented as a set of COM (Common Object model) objects by the server and made available for clients. OPCUA (Unified Architecture) describes the services with WSDL (Web Service Definition Language) instead of the Microsoft specific COM model to make it also available on non Microsoft platforms.

- *OSGI (Open Services Gateway initiative)* [16] is a specification for a dynamic software platform based on the Java virtual machine. Components (called *Bundles*) expose their interfaces (called *Services*) via a service registry. Bundles can be added, updated and removed dynamically at run-time and from remote. OSGI is well suited to residential gateways in home automation, smart grid or assisted living where new services can be easily selected and installed from remote.

1.3 Constrained Restful Environments

REST (Representational State Transfer) is an architecture model based on the World Wide Web following a Request Response pattern. Clients access resources on servers identified via a URI (Uniform Resource identifier) by sending a small number of primitive operations only. These primitives perform the standard CRUD (Create, Read, Update, Delete) functions on the resources. For HTTP the primitives map to the HTTP messages POST, GET, PUT and DELETE. Resources are described by standard notations such as XML or JSON. Compared with RPC (Remote Procedure Call) or SOAP (Simple Object Access Protocol) REST applies simple operations on complex resource descriptions (`GET /users/user1/address`) vs complex operations on simple resource descriptions (`getAddress(user1)`).

Constrained Restful Environments apply the successful REST architecture to IoT gateways and even single IoT devices that make their resources like sensors and actuators available to clients via a web server. These devices are often very small in terms of their processing and memory capabilities which makes it not possible to run a standard HTTP server on them. CoAP (Constrained Application Protocol) [11], like HTTP an application layer protocol, on top of UDP

provides a small footprint alternative for constrained devices. Each device runs a CoAP server that provides clients access to resources. Clients either work directly as a CoAP client or access the CoAP enabled device via HTTP and an HTTP-to-CoAP proxy. Combined with efficient wireless transmission protocols based on IPv6 such as 6LowPAN (IPv6 over Low power Wireless Personal Area Network) and efficient payload encoding techniques such as EXI (Efficient XML Interchange) CoAP provides the base for an IoT architecture where the devices themselves are first class citizens on the network rather than being controlled by a central server application.

1.4 oBIX Standard

oBIX (Open Building Information eXchange) [3] is a standard defined and promoted by OASIS (Organization for the Advancement of Structured Information Standards) that defines an object model and a set of protocol bindings in the domain of building automation systems. oBIX is built on following concepts:

- *Object Model:* defines a set of base objects to handle data from sensors and actuators. The data types comprise boolean, numeric, string and enumeration types to handle points (an abstraction that holds a single scalar and its status and is mapped to sensors, actuators or set points), date and time values to handle histories (sets of timestamped point values) and URIs to name objects.

- *Objects:* describe concrete entities in the physical world (e.g. a lighting controller, an air conditioning system or parts of them). Objects are defined as a set of parts which are either objects from the base object model or objects that itself are composed of objects from the base object model. The definition of objects is recursive, objects can contain other objects.

- *XML:* is used to provide a simple mapping of the object model to a machine and human readable form. Each of the object types maps to one type of XML element, the *val* attribute maps the value of the object. The following sample taken from an object handled by oBIX shows the XML representation. The object *BrightnessActuator* refers to a contract *iot:BrightnessActuator* having one part with the name *value* and a value of 0.

```
<obj href="/BrightnessActuator" is="iot:BrightnessActuator">
  <int name="value" href="value" val="0" writable="true"/>
</obj>
```

- *Encoding:* binary encoding provides a compact representation for usage on low bandwidth connections and constrained devices.

- *Contracts:* are used to define a domain specific object model. They are objects themselves but can be referred to by other objects as a prototype. An object referring to a contract by the *is* attribute ensures that at least all the fields defined in the contract are available in the object, e.g. a certain object *BrightnessActuator* referring to a contract

iot:BrightnessActuator ensures that the *value* which is an integer value is part of it, but other parts like an on/off switch may be added to the object. Contract definitions can be overridden (e.g. defining a new value) but the type defined in the contract can not be changed. Objects can refer to multiple contracts similar to multiple inheritance in object oriented languages. Flattening is used to avoid deep inheritance hierarchies as shown in the following example where *ClockRadio* inherits directly from *Clock*, *Radio* and *Device*.

```
<obj href="Device">...<(obj>
<obj href="Clock" is="Device">
<obj href="Radio" is="Device">
<obj href="ClockRadio" is="Clock Radio Device">
```

- *URIs:* are used to identify objects, in fact the representation of objects in oBIX is a set of XML documents identified by URIs. The usage of URIs as a naming scheme has the advantage that they are already defined in an existing specification (RFC 3986) and programming languages typically provide good support for them.
- *REST (REpresentational State Transfer):* is used as a model to access oBIX objects. Each object identified by an URI is accessible via standard HTTP operations GET, PUT, POST and DELETE mapped to the object operations of read, modify, create and delete.
- *Extendibility:* the concept of objects being recursively composed of other objects and of contracts defined on base of prototype objects provides a high degree of building abstractions without need to touch the basic definition of oBIX. New abstractions are built out of trees of existing objects.

oBIX objects play a server role, they provide means to query their values and set new values but can not send spontaneous messages when a value changes. This is unlike the role of devices typically have in machine-to-machine or home automation systems where they play a client role and can send messages on changed values at any time. oBIX provides the concept of *Watches* to handle rapidly changing object data without requiring the object's client to implement a callback and expose a public IP address. Watches work in following way:

- A *Watch Service* object provides a well-known URI as the factory for creating new watches. When the make operation is performed on a watch service it creates and returns a new *Watch* object.
- The client then adds existing objects to the watch by the add operation.
- In regular intervals the client should call the *pollChanges* operation which returns a list of all objects that have changed since the last call to *pollChanges*.

History objects provide access to timestamped point value data, when ever a value changes, the history object will add the value with the corresponding timestamp to a list of history data. Histories are queried by start time, end time and the maximum number of history entries and return the list of all fitting history records. History rollups provide derivative data from the history

within certain time intervals. A history rollup query delivers the list of derivative data (count, min, max, average and sum of values) for a defined time interval between a start data and an end date to an application interested in the summary values rather than the raw history data.

The *Lobby* object provides access to all objects handled by an oBIX gateway. Utilizing the lobby a client only needs to know the well-known address of the lobby and can retrieve the URIs of all other objects.

1.5 oBIX Gateway

Architecture

The multi protocol IPv6 gateway [12] based on oBIX connects to existing building automation systems (in case of the implementation BACnet and KNX) and makes the devices on these systems available as oBIX objects via a REST or CoAP (Constrained Application Protocol) [4] interfaces. It consists of the following architectural components as shown in the diagram in Fig. 1.2.

- *HTTPHandler:* provides access to the oBIX objects via REST style web services as defined in the oBIX specification. It is based on NanoHTTPD, an open-source, small-footprint web server.

- *CoAPHandler:* provides access to the oBIX objects via the CoAP protocol. It is a mapping of the *PUT, POST and DELETE* messages like Http but in addition provides an observe option to the get message to avoid the polling required by clients to handle oBIX watches. When a GET is combined with *observe* the return message to the get will be sent when ever the status of the object changes. The Californium package by ETH Zurich is used to implement the CoAP protocol handler [14].

- *EXI Parser:* provides an implementation of EXI (Efficient XML Interchange). XML is a powerful and easy to understand protocol binding but adds considerable overhead compared to raw protocols. EXI compresses XML so that also less powerful communications channels and devices can be addressed by XML.

- *oBIX Handler:* provides access to the oBIX objects according to the standard and implements the base objects of the specification such as int, bool, real, str as well as the lobby object, watches, lists and histories including rollup.

- *IoT Objects:* is a mediation layer that provides access to non oBIX devices through the oBIX handler. Such objects will be connected to the oBIX layer through a set of adapters. The KNX adapter and the BACnet adapter handle the corresponding protocols. The virtual device adapter provides a set of virtual devices that can be used for testing purposes without a set of real devices behind. Virtual devices adhere to the same interface as real devices and behave exactly in the same manner.

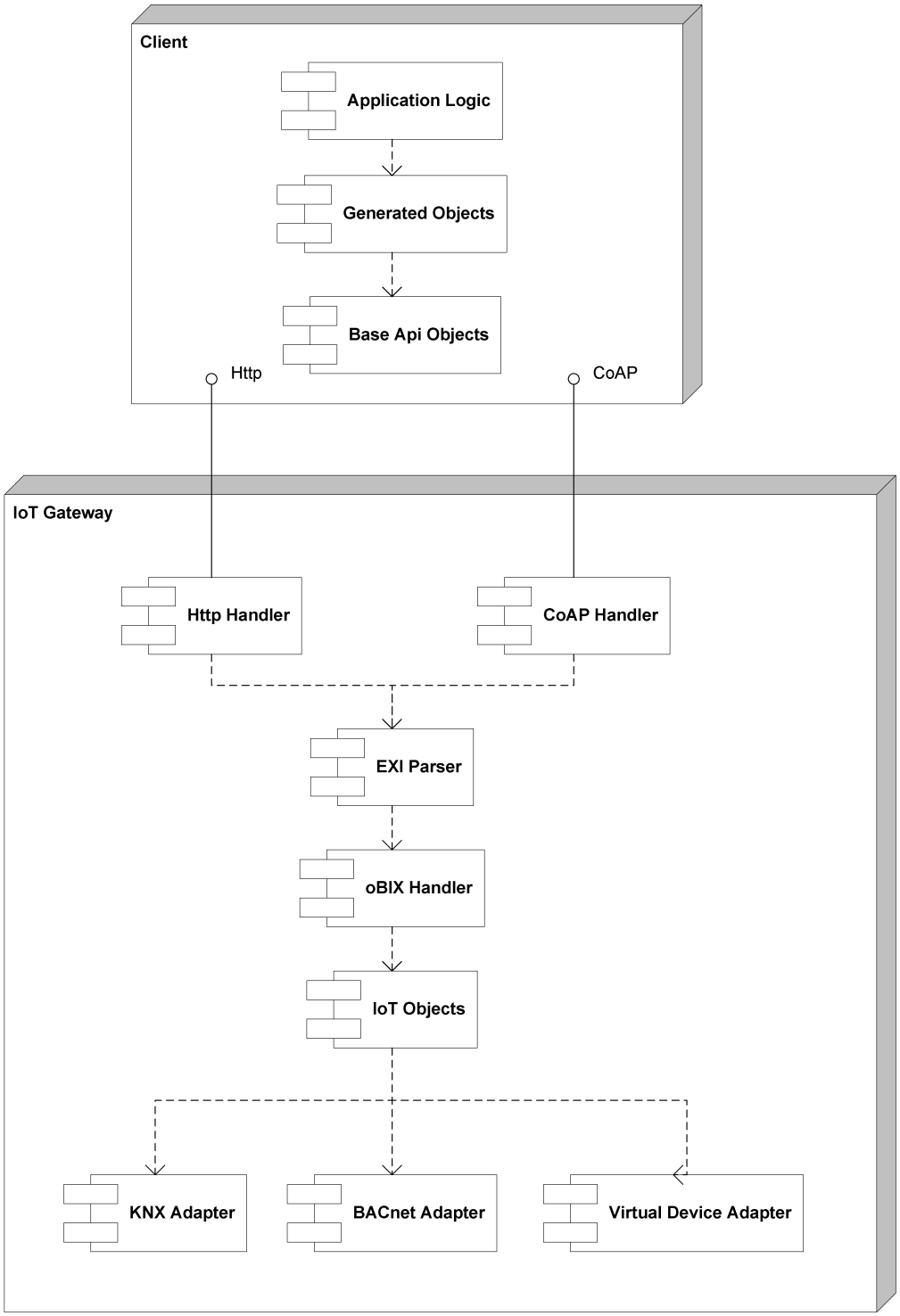

Figure 1.2: Multi Protocol Gateway and API Components

Implementation

From the implementation point of view packages comprising the multi protocol gateway are:

- The `obix.server` package contains the CoAP server and the HTTP server
- The `obix.server.objects.iot.actuators.*` and the `obix.server.objects.iot.sensors.*` packages contain the interface definitions as well as the implementation of KNX, BACnet and virtual devices
- The `obix.objects` package contains the standard oBIX base objects such as base types, watch and lobby
- The `californium.*` packages contains the sources of the CoAP implementation of ETH Zurich [14]

Usage

The oBIX objects handled by the multi protocol gateway can be accessed by an HTTP REST style client or by a CoAP client. For HTTP testing purposes HTTP Requester, a plug-in for Mozilla Firefox is well suited, for CoAP access Copper, which is also a Firefox plug-in can be used. While testing is quite easy to achieve writing clients embedded in an application requires handling a lot of non-business protocol handling logic in each client. The approach outlined in the next chapter will describe an API providing a convenient way to embed usage of the gateway in client code.

1.6 oBIX Toolkit

Summarizing the documentation accompanying the Java oBIX Toolkit, it is an open source, public domain library to support oBIX developers. Toolkit functions that will be used in the implementation of the domain specific language are:

- The obix package provides a set of classes for modelling each of the oBIX built-in object types: `Obj`, `Bool`, `Int`, etc. These classes provide support for managing tree structures, contract definitions, and encoding/decoding of primitive data types
- The `obix.io.ObixEncoder` class takes an object tree and generates its XML representation. Likewise the `obix.io.ObixDecoder` parses an XML representation into memory as a tree of object instances. The `obix.io.BinObixEncoder` and the `obix.io.BinObixDecoder` are used for binary serialization. Encoder and decoder will be used in the implementation of the domain specific language.
- The oBIX compiler is a tool which reads an oBIX XML document and outputs Java source code - one interface for each contract. The compiler will be enhanced with the ability to generate also implementation classes in the course of the development of the API.

1.7 Problem Statement

Clients access the oBIX objects via HTTP or CoAP protocol. Whereas such an access is not hard to implement, especially from web based applications, it requires the application developer to handle the protocol specific implementation, message handling and payload settings.

- Implementation of protocol management code has to be repeated for each new application. There is a lot of code to write that has nothing to do with the business logic of the application.
- The application is bound to a specific protocol (CoAP or HTTP) and it is not possible to switch between them easily.
- To support different encodings would need a specific implementation in every new application. Applications can not switch between encodings or decide on encoding based on configuration.
- Applications have to care about updating object state by applying the watch or observer pattern and handle repeated polling for the watch pattern and asynchronous responses for the observer pattern.
- Text level access to objects inhibits standard type-safe access provided by object oriented programming.

Combined together this makes writing even a simple application quite a difficult task. The following example summarized pseudocode in Listing 1.1 takes 110 lines of code to realize a simple temperature alarm application which monitors a temperature and switches on an alarm light if the temperature is below or above a certain value. The business logic of this application is less than 20 lines of code.

Listing 1.1: Temperature Alarm

```
public class TemperatureAlarm {
  set alarm = false;

  set maxTemp = 28.5;

  public static void main(String[] args) {
    set uri = "coap://[2001:620:2080:130::c8]/temperature";
    assign new GETRequest
    specify URI of target endpoint
    enable response queue
    set observe option
    set media type = APPLICATION_XML

    execute the request
```

```
    wait to receive the response

    set val = extract value from payload
    printout val

    register response handler with {
      public void handleResponse(Response response) {
            set temp = extract value from payload
            if(temp < maxTemp && alarm ) {
              assign new PUTRequest
              set target URI of request
              set payload of request (false)
              execute request
              set alarm = false
              }
              if(temp > maxTemp && !alarm ) {
                assign new PUTRequest
                set target URI of request
                set payload of request (true)
                execute request
                set alarm = true
              }
      }
    }
    wait on termination
  }
}
```

1.8 Aim

It is aim of the thesis to provide a simple, type-safe business oriented API that encapsulates all the protocol specific functionality in basic classes and provides an API to client applications that just requires the client to instantiate proxy objects and manipulate them. The proxy objects cover then communication, encoding and protocol specific patterns. The evaluation in Section 3 of the thesis will compare the client application based on the domain specific API with the plain protocol based client application. The API follows a similar pattern as wsdl2java which takes an XML description of a web service and generates Java client stubs that can be used by the application developer like local objects.

CHAPTER 2

Design and Implementation

2.1 Overview

The design of the domain specific API covers

- definition of an object model as base for device abstraction
- deriving a set of oBIX contracts implementing the object model
- using an existing toolkit (oBIX toolkit) to build a set of Java classes implementing the oBIX contracts and adding HTTP and CoAP client code to these Java classes to implement access to the real IoT devices
- building a domain specific language based on Groovy on top of the Java classes to facilitate simpler access to the library's functionality

Object Model

The object model covers the capabilities of devices used in home and building automation but avoids the full complexity possible with arbitrary oBIX contracts and provides:

- a *device* represents a physical device in the domain. It consists of a set of sensors and actuators and a set of configuration parameters
- a *sensor* is part of a device and is capable to handle values measured in its environment,like a temperature or a position. Each sensor is capable to handle a set of attributes, like a position sensor that might have longitude, latitude and altitude as attributes or a temperature sensor that might have the value in degrees Celsius and the accuracy as attributes. Measurements taken contain the actual value and a timestamp when the value was measured.

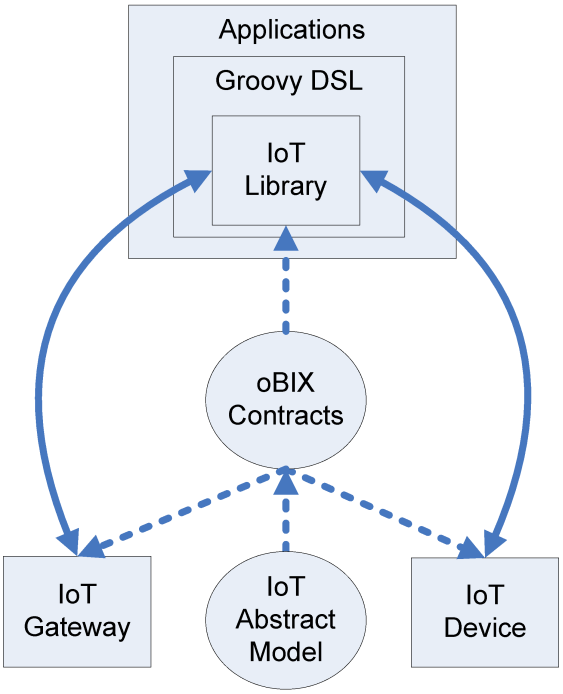

Figure 2.1: Architecture

- an *Actuator* represents a command that the device can perform, e.g. to set a valve. Each command can have a number of attributes that control the command, e.g. the desired position of the valve.

oBIX mapping

The mapping is straight forward. Each of the abstract model elements is represented by an oBIX contract, specific devices, sensors and actuators are then represented by oBIX contracts inheriting from the abstract contracts.

oBIX toolkit

The idea is that the toolkit builds automatically a set of Groovy classes from the set of concrete defined oBIX contracts. In this way the library becomes independent of the actual set of defined oBIX contracts as long as they are based on the same abstract model. The generated classes are then enhanced with code to communicate with the device via HTTP and CoAP. This is realized both as an add on to the oBIX toolkit but also requires modification of the toolkit itself.

Groovy DSL

The final step is to build a DSL (domain specific language) based on Groovy on top of the API. This DSL provides access to the IoT objects in an easy to use manner. Compared with

a Java application that uses the web services directly the number of lines of code to write an application is greatly reduced. The final API can then be integrated into J2EE applications or into cloud based application frameworks such as Cosm [5] or M2MLabs Mainspring framework [6] to be deployed on public or private clouds, but this is not part of the bachelor work.

2.2 Requirements

The API should fulfil the following basic requirements:

- *Protocol independence:* The API should provide a view to the client that is the same, independent of the underlying communication protocol (HTTP or CoAP).
- *Communication agnostic:* The client should not be aware of communication issues, the API objects should work as local proxies working in a synchronous manner.
- *Support for arbitrary contracts:* The API has to support an arbitrary set of oBIX contracts not known at API building time. The API is restricted to objects implementing a list of contracts, objects not implementing a contract are not supported.
- *Type safe interface:* The API objects should provide a type safe interface, only methods should be callable that are defined in the contracts implemented by the object. Type checking should be done preferred at compile time.
- *Support for encoding:* All encodings provided by the IoT multi protocol gateway (XML, JSON, Octet-Stream, Binary, EXI) have to be supported by the API.
- *Watches and Observers:* oBIX watches and CoAP observer functionalities have to be supported. Usage of watches and observers should be chosen automatically based on the protocol.
- *Histories:* oBIX histories including roll-up and queries have to be supported by the API. An oBIX object having a history defined should get automatically in the API corresponding methods to handle the history.
- *Lobby access:* should be provided independent of the communication protocol used. The lobby should give access to all objects handled by the multi protocol gateway.

2.3 Code Generation Approach

All oBIX contracts are dynamic in the sense that a contract is just a plain oBIX object that is used as reference in an *is* attribute. It is possible to add a new piece of equipment (e.g. an HVAC controller) to the set of oBIX contracts at any time and then create oBIX objects referencing this contract. There a three possible approaches to deal with the dynamics of the oBIX contracts:

- *Generic Methods:* We just define generic methods with a text based interface that access oBIX objects and their parts via names. Advantage is that it is very simple to implement, disadvantage is that there is no type-safe interface, wrong names are only detected at runtime, Java generics can not be used for type-safe handling in application code.

- *Code Generation:* API objects are generated on-demand by reading a set of oBIX contracts and generating Groovy code from the contracts. Advantage is a full type-safe interface fulfilling all requirements, disadvantage is the complexity of oBIX contract parsing and code generation. The oBIX toolkit helps a lot here providing a full oBIX object (and therefore also contract) parser and code generation capability for Java interfaces. The code generation capability will be enhanced in the course of this work to generate full Groovy implementation classes from existing contracts.

- *Groovy Dynamic Methods:* An alternative approach is to use the dynamic methods of Groovy for API object implementation. Dynamic objects allow a type-safe usage but exceptions due to unknown methods in objects are only detectable at runtime.

In the implementation of the API the code generation approach will be used with Groovy dynamic methods as a parallel alternative way.

2.4 API Classes Static View

The UML class diagram (see Fig.2.2) gives a static view on the implementation classes of the API. All generated objects (marked in grey colour) are derived from a *BaseObj* that contains

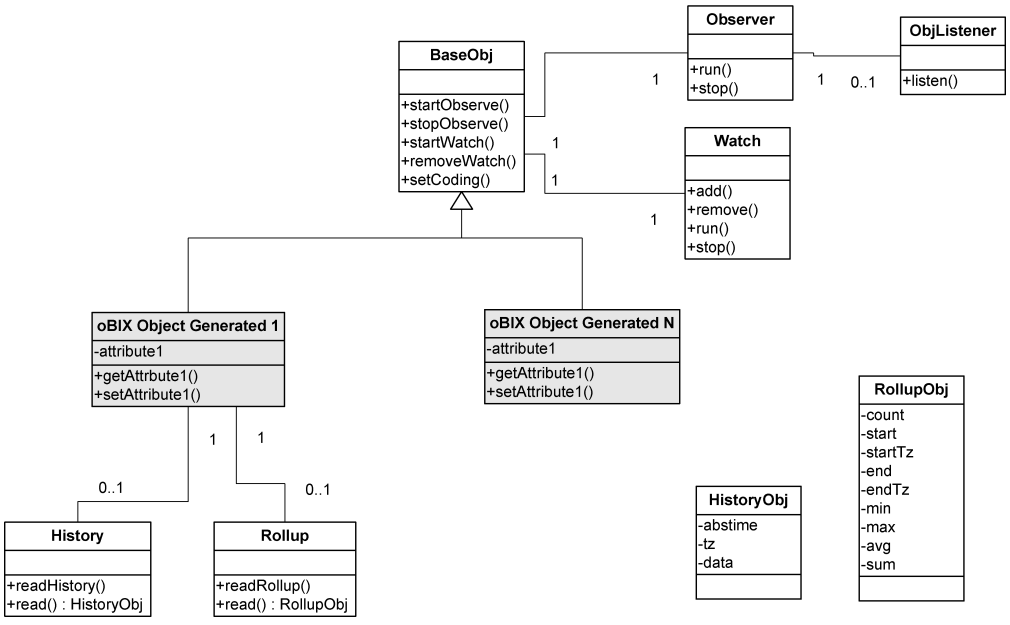

Figure 2.2: Static View

all generic functionality so that generation only adds the type specific methods according to the oBIX contract. The base object includes a *Watch* and an *Observer* object that provide base functionality for the corresponding oBIX objects. Each of the generated objects may include a *History* and a *Rollup* object depending on the oBIX contract characteristics. *ObjListener* is an interface that contains the listen method which is called when an observed object's value changes. Clients have to provide application specific classes that implement ObjListener in order to use this functionality.

2.5 API Classes Dynamic View

Reading Objects

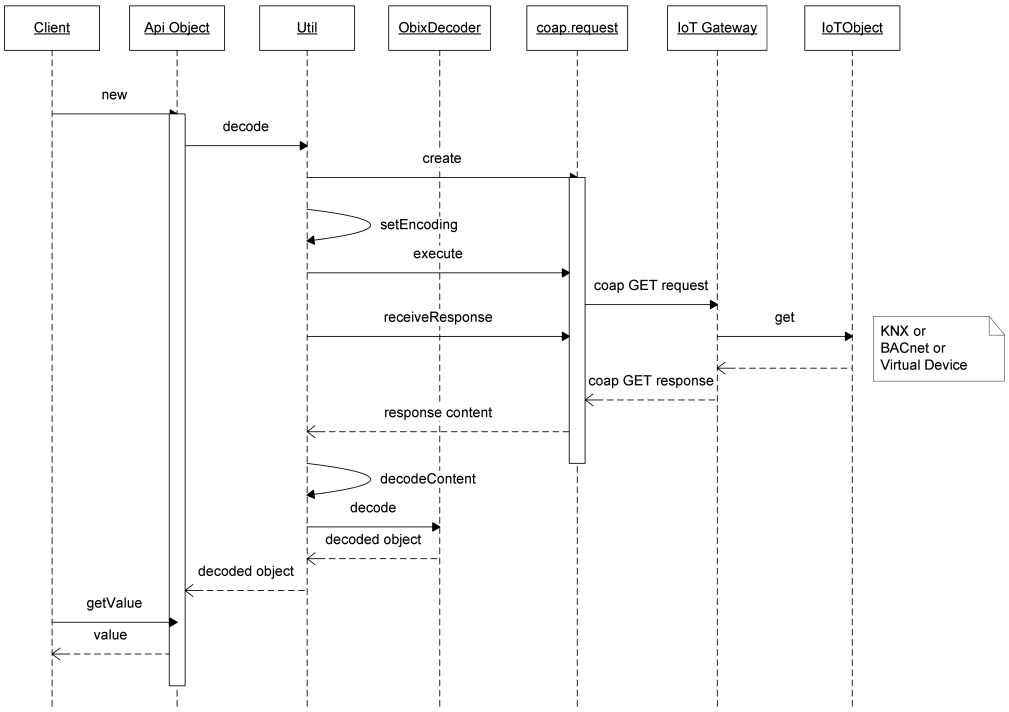

Figure 2.3: Reading Objects

When a client creates a new API object the static method *decode* is called which performs the following steps:

- Create a CoAP GET request (as shown in (Fig. 2.3) or an HTTP GET request depending whether the URI starts with CoAP or HTTP.
- Set the encoding as defined in a parameter of the object constructor.
- Execute the request and wait for the response from the gateway.

- Decode the content and return an oBIX toolkit *Obj* object.
- The generated API class then provides type-safe field access methods to read the API object. Subsequent reads of fields will not trigger an update of the object's content. To update the content the watch or observation mechanism has to be used or the object is updated by re-creating it with new(). This implementation has been chosen to avoid performance issues due to frequent reading from gateway.

Writing Objects

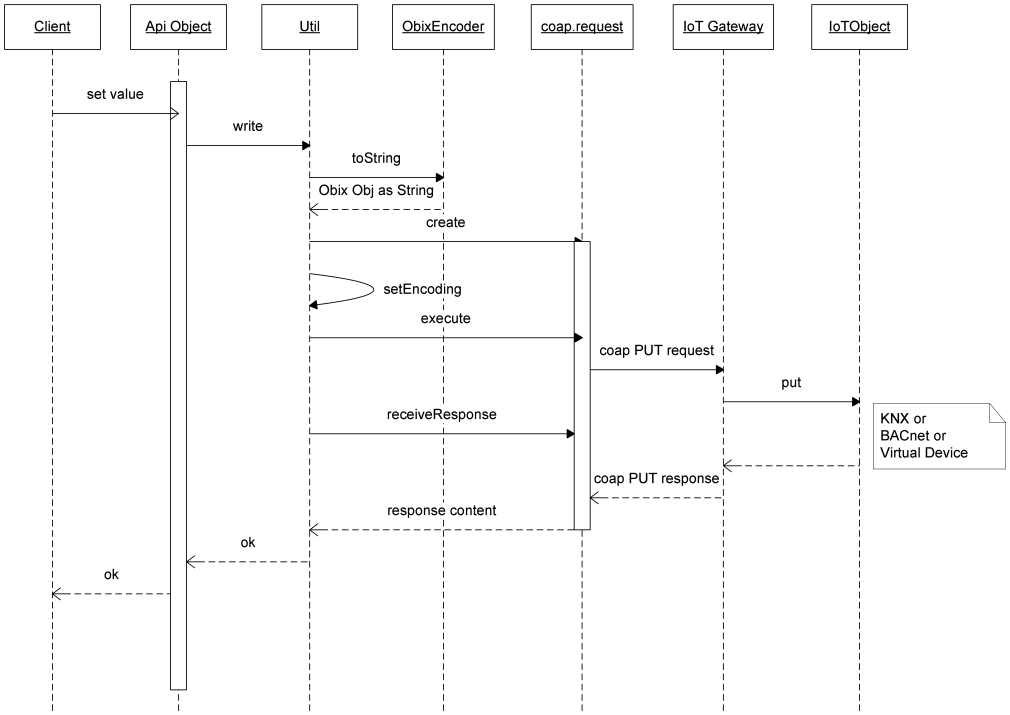

Figure 2.4: Writing Objects

When a client writes an API object (as shown in Fig. 2.4) the static method *write* is called which performs the following steps:

- Update the internal oBIX toolkit *Obj* object and encode it as a String using ObixEncoder.
- Create a CoAP PUT request (as shown in the diagram) or an HTTP PUT request depending whether the URI starts with coap or http.
- Set the encoding as defined in a parameter of the object constructor.
- Execute the request and wait for the response from the gateway.

- Check the response and return the result. Although the gateway returns a complete object as response only the result is checked here because the API object has already been updated locally.

- It is assumed here that writeable fields of the IoT object are updated only by clients and not by internal changes in the IoT object. In such a case race conditions could occur that could overwrite an internal state change by a client state change and transactional writes with a locking mechanism would be required.

Watch

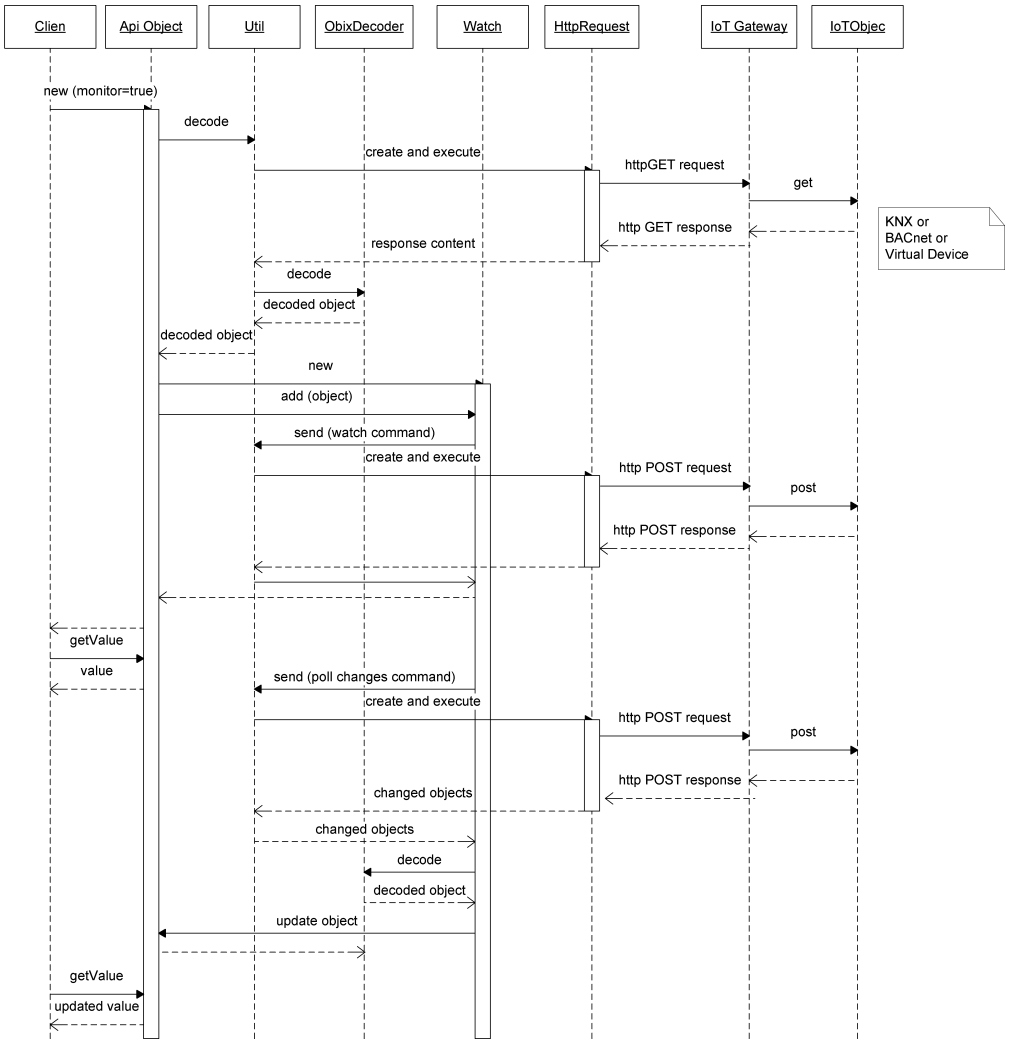

Figure 2.5: Watch

A *Watch* is created when an API object is created that defines monitor=true in a parameter of its constructor (see Fig. 2.5). Depending on the type of URI (either http or coap) a *Watch* is created for HTTP and an *Observer* is created for CoAP. The following steps are performed:

- A new API *Watch* object is created that is attached to the object. The watch is then added to a watch object in the IoT gateway via a POST message.
- The API *Watch* object periodically in an own thread issues a *pollForChanges* message to the gateway via a POST message. The gateway will respond with an updated object if the object in the IoT gateway has changed since the last *pollForChanges* command.
- The API *Watch* object then identifies the corresponding API object and updates its status.
- All further reads of the API object will return the new contents.

As shown in the state diagram (Figure 2.6) a *Watch* object is created either in monitored state or in not monitored state. At any time during the lifetime of the object it is possible to switch between these states by the add() and remove() methods.

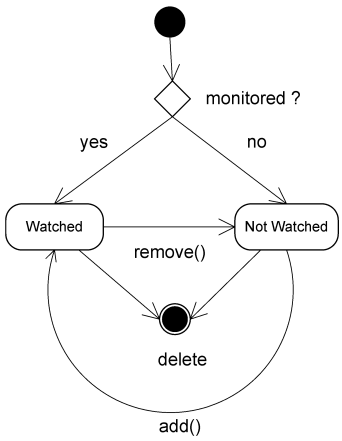

Figure 2.6: Watch States

Observer

An *Observer* is created when an API object is created that defines true for the monitor parameter of its constructor (see Fig. 2.7) Depending on the type of URI (either http or coap) a Watch is created for HTTP and an Observer is created for CoAP. The following steps are performed:

- A new API *Observer* is created that is attached to the object. The observer sends a CoAP GET request to the gateway to enable observation.
- The gateway will send an unsolicited response whenever the observed IoT object changes.

- The API *Observer* then identifies the corresponding API object and updates it. Optional, if an object implementing the *ObjListener* interface is supplied in the constructor, the listen method of this object is called. *ObjListener* allows to implement application specific code that has to be executed whenever an IoT object changes.
- All further reads of the API object will return the new content.

As shown in the state diagram (see Fig. 2.8) an object is created either in monitored state or in not monitored state. At any time during the lifetime of the object it is possible to switch between these states by the start() and stop() methods.

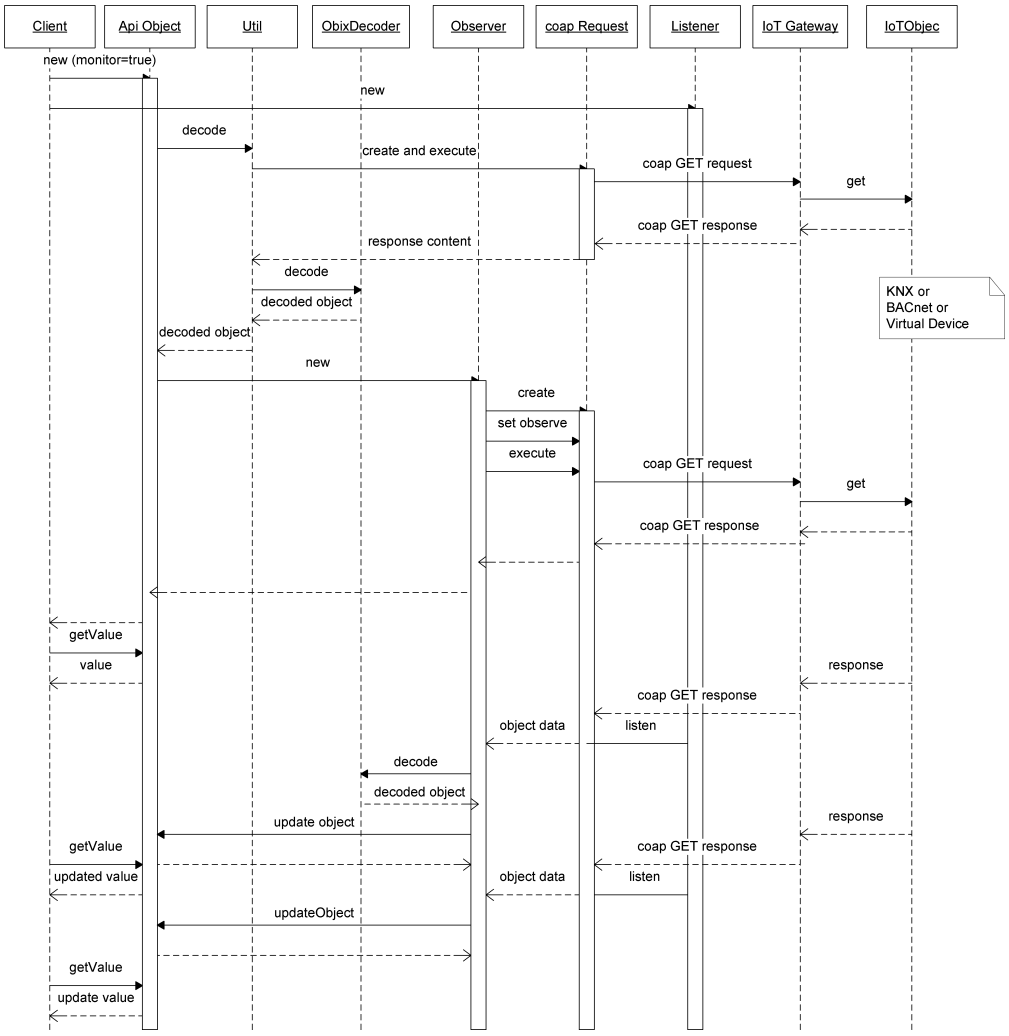

Figure 2.7: Observer

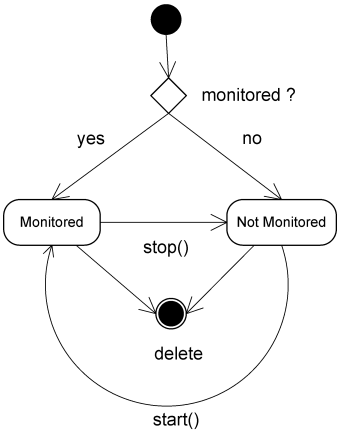

Figure 2.8: Observer States

History

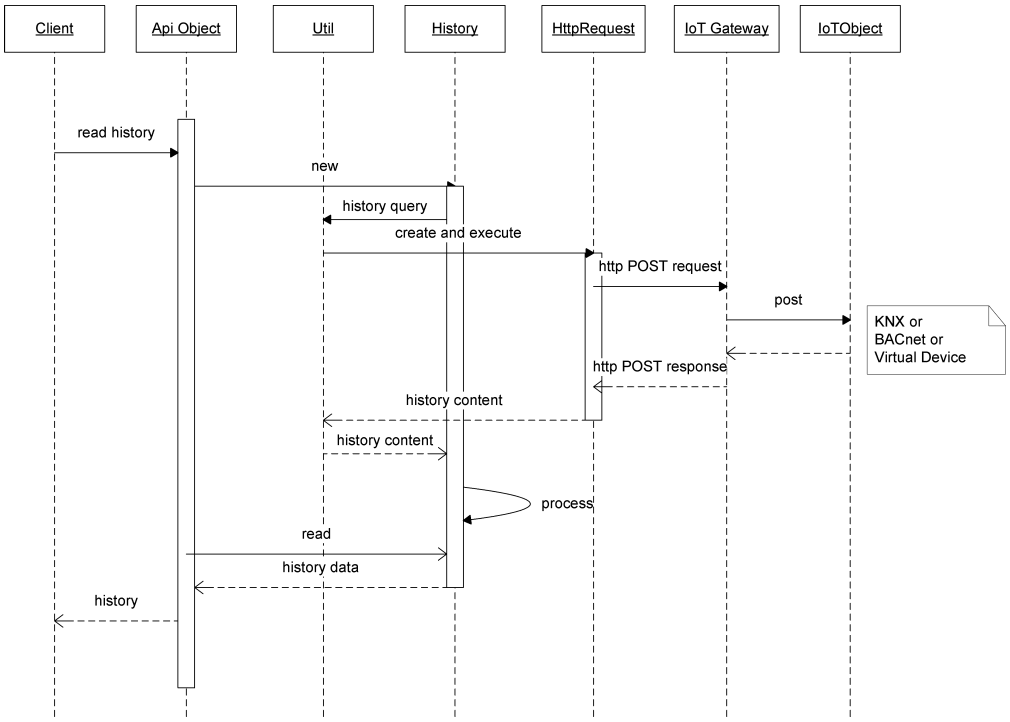

Figure 2.9: History

A new *History* is created when a client calls the read history method of the API object, the method is generated when the oBIX contract includes a part that refers to the obix:History in its contract (see Fig. 2.9). The name of the method is composed of <*field name*>*History*. The

following steps are performed when reading a history:

- When the *History* is created a history query is sent via a POST message to the IoT gateway. The gateway responds with a list of timestamped values of the element requested.
- The *History* processes the results and stores them in a list of *HistoryObj* objects.
- The client can then retrieve the results via the read history method.

History Rollup

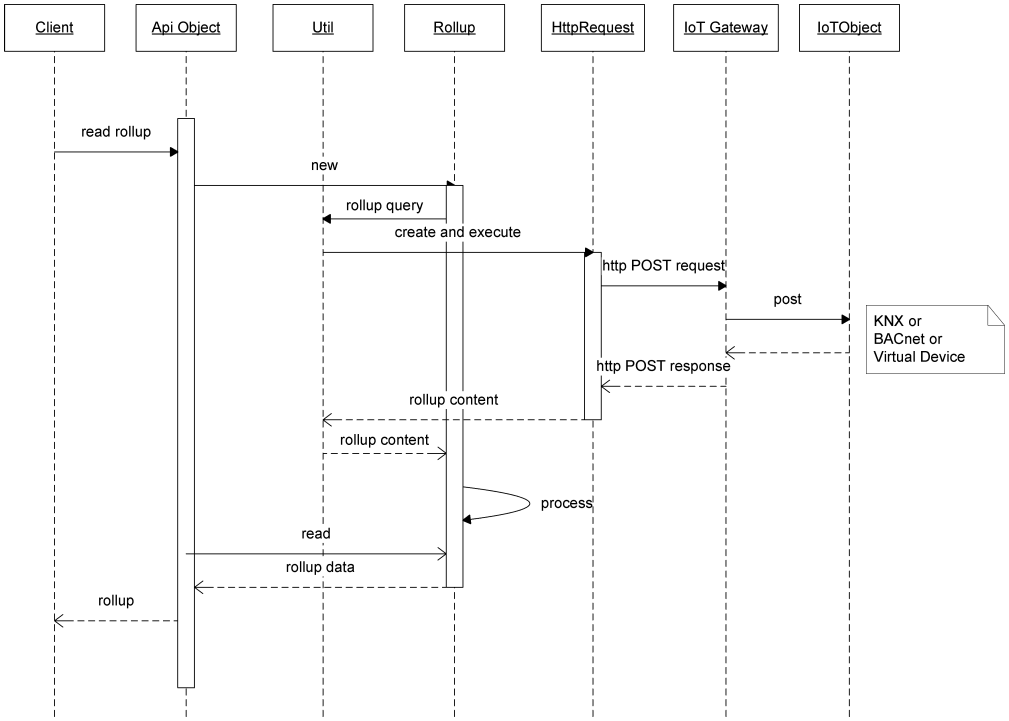

Figure 2.10: History Rollup

A new *Rollup* is created when a client calls the read rollup method of the API object, the method is generated when the oBIX contract includes a part that refers to the obix:History in its contract (See Fig. 2.10). The name of the method is composed of <*field name*>*Rollup*. The following steps are performed when reading a history rollup:

- When the *Rollup* is created a rollup query is sent via a POST message to the IoT gateway. The gateway responds with a list of timestamped rollup values of the element requested.
- The *Rollup* processes the results and stores them in a list of *RollupObj* objects.
- The client can then retrieve the results via the read rollup method.

Lobby

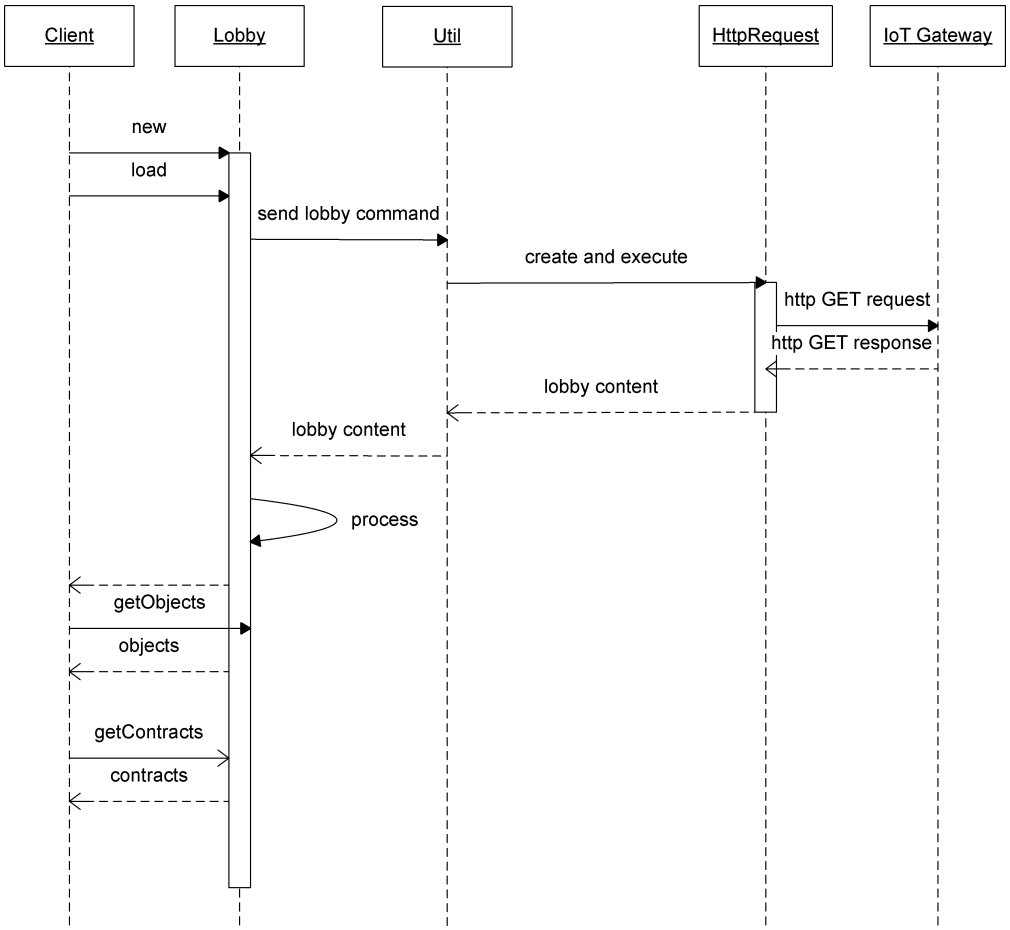

Figure 2.11: Lobby

The *Lobby* allows access to the set of objects managed by the IoT gateway (see Fig. 2.11). It provides a method to retrieve the URIs of all objects as a list and a method to get a list of all contracts implemented by a certain object.

2.6 API Class Generation

Obixc

The oBIX compiler provides a class that reads a set of strings (either oBIX URIs or file names) in its main method and generates a set of Java interfaces for each oBIX object and for the object's children recursively. The compile method uses the ObixDecoder to parse the oBIX object into a tree representation and then maps the oBIX well known types to the tree nodes. These well known types are realized as a set of classes in the oBIX toolkit. Finally, a simple text based

generation procedure walks over the tree and generates the interface classes. All information required for generation such as the names of oBIX objects and their types are contained in the parse tree.

Modifications for the oBIX API

These modifications comprise:

- generation of Groovy API class files to the compile() method. Groovy compilation is done when the command line arguments contain the argument *groovy*.
- the ability to generate interfaces is not used in the API and the ability to generate classes recursively for oBIX objects is disabled because we do not want that own API classes are generated for an object's fields but rather a method in the parent class.
- the *writeImplementation()* method was added. This method generates the API classes. Listing 2.1 shows the modified compile() method.)

Listing 2.1: Compile Method

```
public void compile(InputStream in)
  throws Exception
{
  // decode document
  ObixDecoder decoder = new ObixDecoder(in);
  decoder.setUseContracts(false);
  Obj root = decoder.decodeDocument();

  // map well known objects to types
  map(root);

  // compile the well known types to .java or .groovy files
  if (language.equals("groovy")){
      writeImplementation(types[0]);
  } else {
      for (int i=0; i<types.length; ++i)
        writeInterface(types[i]);
      // write ContractInit.java
      writeContractInit();
  }
}
```

A simple string based approach is applied when walking on the object tree and writing the corresponding Groovy classes. It was also investigated to use a template based approach but it was discarded because of additional implementation complexity. Figure 2.12 shows the class

generation algorithm. Listing 2.3 shows the generated class for a simple contract of an object handled by the IoT gateway. The generated code is simple, all code parts that are not dependent on the specific contract are handled by the API BaseObj class. All generated classes inherit from BaseObj.

Listing 2.2: Generated Class

```
package at.ac.tuwien.auto.iotsys.api.generated;

import obix.*;
import at.ac.tuwien.auto.iotsys.api.*;import java.util.List;
import at.ac.tuwien.auto.iotsys.api.Util.Coding;


/**
 * IotBrightnessActuator
 *
 * @author     obix.tools.Obixc
 * @creation   02 Mrz 13
 * @version    $Revision$ $Date$
 */

public class IotBrightnessActuatorImpl extends BaseObj
{
  public IotBrightnessActuatorImpl(String urlString, Coding
     coding, boolean monitor, ObjListener ol) {
     super(urlString, coding, monitor, ol)
  }

  public IotBrightnessActuatorImpl(String urlString) {
     super(urlString, false, false, null)
  }

  public Int value(){
     return (Int)decodedObj.get("value")
  }

  public void value(Int value){
      Int obj = (Int)decodedObj.get("value")
      obj.set(value.get())
      Util.write(urlString, decodedObj, coding)
  }
```

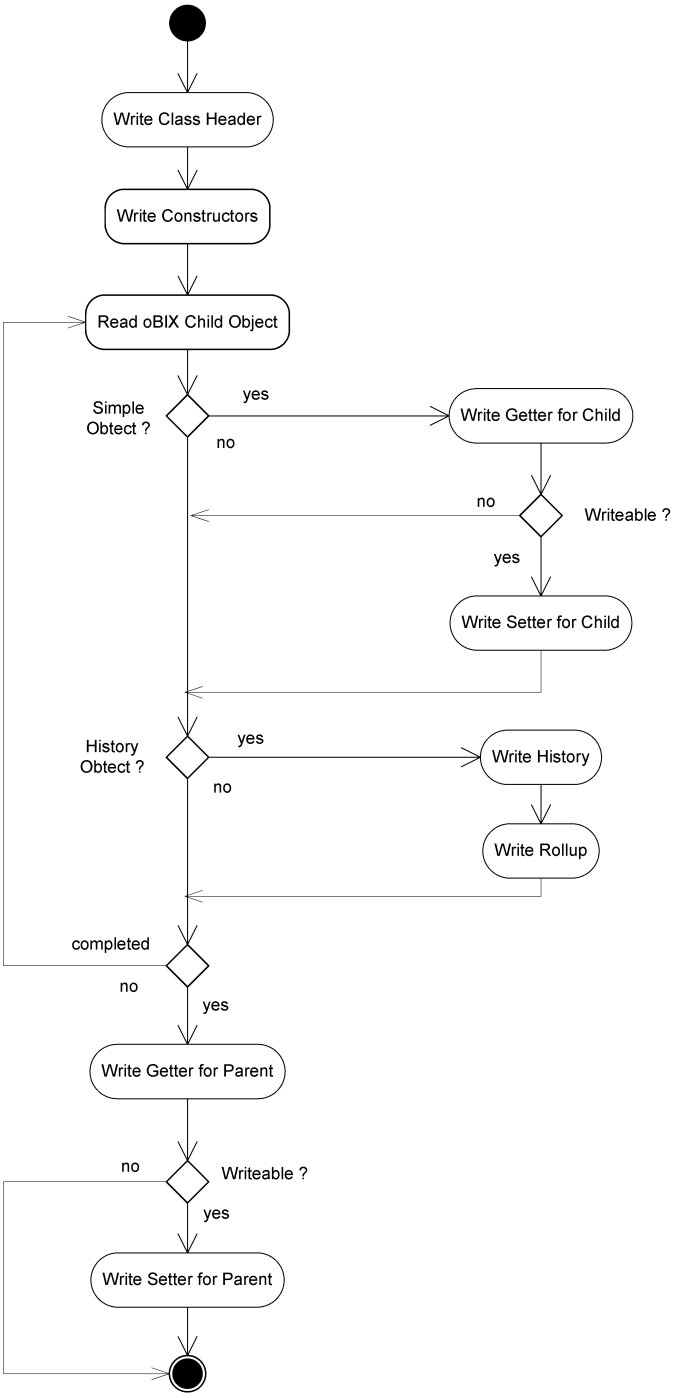

Figure 2.12: Compilation Process

```
  public List<HistoryObj<Int>> valueHistory(int limit, String
     from, String to){
      History<Int> history = new History<Int>(urlString, "value
         ", "Int", coding)
      return history.read({ new Int() }, limit, from, to)
  }

  public List<RollupObj> valueRollup(int limit, String from,
     String to, String interval){
      Rollup rollup = new Rollup(urlString, "value", coding)
      return rollup.read(limit, from, to, interval)
  }

  public Obj getIotBrightnessActuator(){
     return(Obj)decodedObj
  }
}
```

CHAPTER 3

Evaluation

3.1 API Usage

Class Generation

Classes are generated by the modified obixc compiler based on contracts.
Let's take the *BrightnessActuator* object as example. Using HttpRequester we retrieve

```
<obj href="/BrightnessActuator" is="iot:BrightnessActuator">
  <int name="value" href="value" val="20" writable="true"/>
  <ref name="value history" href="value/history" is="obix:History"/>
</obj>
```

By replacing the *href* with a contract name (which may not contain special characters) we get a contract definition that is valid for API class generation. The contract reference in the history ref part has to be kept, based on this information the generator will include a history retrieval function in the generated class. Taking the example we get following contract:

```
<obj href="IotBrightnessActuator" is="iot:BrightnessActuator">
  <int name="value" href="value" val="0" writable="true"/>
  <ref name="value history" href="value/history" is="obix:History"/>
</obj>
```

To generate the class we call the obixc compiler with the batch script genclass.bat:
`genclass.bat <contract file> <target directory> <package name>`
or directly with:
`java -cp ../obix-toolkit/dist/obix.jar obix.tools.Obixc <contract file> <target directory> <package name> groovy`

The generated classes can then be used together with the base API obix-api.jar in applications. obix-api.jar can be generated from the source with ant jar.

Creating objects, reading and writing

Creating objects, reading and writing is straight forward like for any Java POJO, see following code snippet

```
def dev = new IotBrightnessActuatorImpl(url,coding,false,null)
// read a field
System.out.println("value=" + dev.value())
// write a field
dev.value (new Int (100L))
```

Arguments for object creation are:

Parameter	Description
uri	the object URI as defined in the IoT gateway, either as HTTP or as CoAP URI
coding	XML | SON | EXI | OCTET-STREAM | X-OBIX-BINARY
monitor	if true monitoring is enabled via observe for CoAP or watch for HTTP
ol	object implementing the ObjListener interface

Lobby

The *Lobby* provides access to a list of objects and for each object a list of contracts, see following code snippet

```
Lobby lobby = new Lobby(url, coding)
List<String> objects = lobby.getObjects()
Iterator<String> it1 = objects.iterator()
boolean objectFound = false
while (it1.hasNext()){
   String href = it1.next()
   println("href="+ href)
   List<String> contracts = lobby.getContracts(href)
   Iterator<String> it2 = contracts.iterator()
   while(it2.hasNext()){
      String contract = it2.next()
      println("contract="+ contract)
   }
}
```

Arguments for lobby construction are:

Parameter	Description
uri	the object URI as defined in the IoT gateway, either as HTTP or as CoAP URI
coding	`XML | JSON | EXI | OCTET_STREAM | X_OBIX_BINARY`

History

The *History* provides access to timestamped historic values of objects, see following code snippet:

```
def result = dev.valueHistory(2,
   "2013-02-18T21:50:50.171+01:00",
   "2013-02-28T21:50:50.171+01:00")
result.each {res ->
   System.out.println("abstime=" + res.abstime)
   System.out.println("tz=" + res.tz)
   System.out.println("value=" + res.data)
}
```

Arguments for history retrieval are:

Parameter	Description
limit	maximum number of results retrieved, 0 means no limit
from	date as `yyyy-MM-ddTHH:mm:ss.SSS+TZ`
to	date as `yyyy-MM-ddTHH:mm:ss.SSS+TZ`

History Rollup

The history rollup provides access to summarized timestamped historic values of objects, see following code snippet:

```
def results = dev.valueRollup(2,
   "2013-02-18T21:50:50.171+01:00",
   "2013-02-28T21:50:50.171+01:00",
   "PT1S")
results.each {res ->
   System .out.println("count=" + res.count)
   System .out.println("start=" + res.start)
   System .out.println("startTz=" + res.startTz)
   System .out.println("end=" + res.end)
   System .out.println("endTz=" + res.endTz)
   System .out.println("min=" + res.min)
   System .out.println("max=" + res.max)
   System .out.println("avg=" + res.avg)
   System .out.println("sum=" + res.sum)
}
```

Arguments for history retrieval are:

Parameter	Description
limit	maximum number of results retrieved, 0 means no limit
from	date as `yyyy-MM-ddTHH:mm:ss.SSS+TZ`
to	date as `yyyy-MM-ddTHH:mm:ss.SSS+TZ`
interval	specification string for time intervals as defined in oBIX specification

3.2 Comparison

Listing 3.1 shows an implementation of the temperature alarm application based on usage of the Groovy API.

Listing 3.1: Temperature Alarm

```
class TemperatureAlarmByApi implements ObjListener {

  boolean alarm = false;
  final double maxTemp = 28.5d;
  IotLightSwitchActuatorImpl lightSwitch
  IotTempControlImpl tempControl

  public void start(){
    lightSwitch = new IotLightSwitchActuatorImpl("coap://
        localhost:5683/virtualLight1",Coding.XML,false,null)
    tempControl = new IotTempControlImpl("coap://localhost
        :5683/tempControl1",Coding.XML,true,this)
  }

  public static void main(String[] args) {
    TemperatureAlarmByApi controller = new
        TemperatureAlarmByApi()
    controller.start();
    while(true){
      Timer.sleep(1000)
    }
  }

  private void checkAlarm(double temp){
    if(temp < maxTemp && alarm ){
      System.out.println("min alarm")
      lightSwitch.value (new Bool(false))
      alarm = false;
    }
    if(temp > maxTemp && !alarm){
      System.out.println("temp alarm")
      lightSwitch.value (new Bool(true))
```

```
        alarm = true;
      }
    }

    public void listen(BaseObj obj){
      IotTempControlImpl tempControl = (IotTempControlImpl)obj
      double temp = tempControl.temperature().getReal()
      System.out.println("Temperature is " + temp)
      checkAlarm(temp);
    }
}
```

Compared to the protocol based implementation, see Listing 1.1, the Groovy API based implementation provides full type-safe access to the oBIX objects, works independent of the used protocol (HTTP or CoAP) and requires significantly less code to implement the functionality.

Technology	Type-safe	Protocol Independent	Lines of Code
Pure CoAP	-	-	110
Groovy API	X	X	35

Table 3.1: Compare Groovy DSL to plain CoAP

3.3 Integration in Server based Frameworks

IoT frameworks or machine-to-machine application platforms like Cosm or M2MLabs Mainspring typically work in a server mode where the central system is receiving messages, sending commands and storing device data. The server would wait for incoming messages over HTTP and not be able to poll regularly for the object status or support CoAP with its observe mechanism directly. This is contrary to the oBIX specification where the oBIX objects themselves work in the server role. A solution is to use the ObjListener concept of the oBIX API for the device status whereas device commands can be sent directly. The diagram in Fig. 3.1 shows the approach.

3.4 Alternative Approach without Code Generation

The implementation language Groovy which extends Java with concepts of dynamic languages enables an alternative approach to code generation. Using meta-programming it is possible to write handlers for missing methods that perform the required functionality. In case of the oBIX API library we can write a generic handler for the setter and getter methods of API objects instead of generating the implementation. Listing 3.2 shows the generic API class.

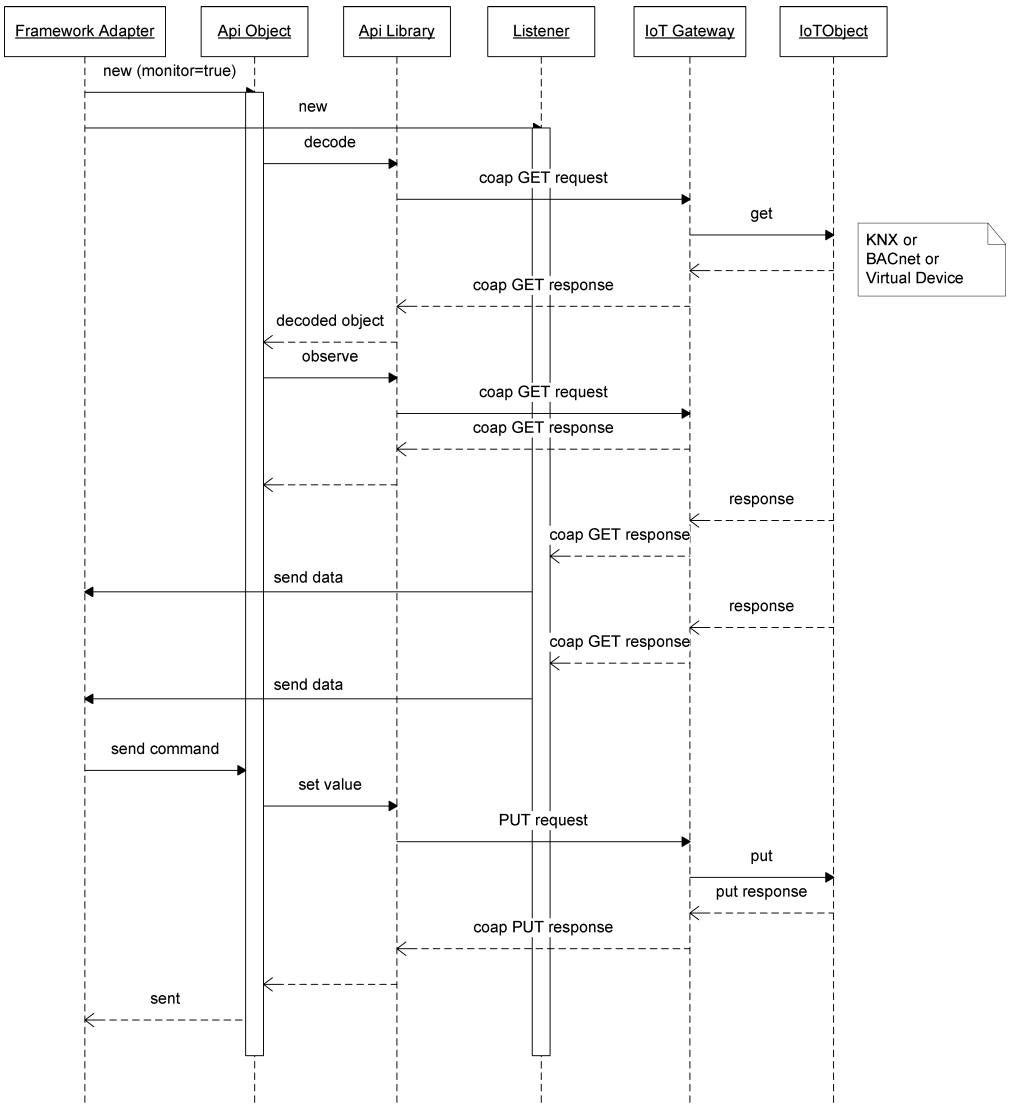

Figure 3.1: IoTFramework

3.5 Handling Objects with Multiple Contracts

A further advantage of the meta-programming approach is that this works also if an oBIX object implements several contracts. Since the underlying oBIX toolkit Obj is able to handle such cases and also the dynamic methods can handle them it is only required to create one API object for all contracts. In case of the generation approach each of the contracts will have its own corresponding API class and the client must instantiate API objects with the same URI separately for each of them.

Listing 3.2: Generic API Object

```
public class ObixAPIObj extends BaseObj {

  public ObixAPIObj(String urlString, Coding coding, boolean
     monitor, ObjListener ol){
    super(urlString, coding, monitor, ol)
  }

  def methodMissing(String name, args){
    String method = "$name"
    if (method.endsWith("History")){
      String objectName = method.substring(0, method.length() -
          7)
      History<Obj> history = new History<Obj>(urlString,
         objectName, "Obj", coding)
      return history.read(args[0], args[1], args[2])
    }
    if (method.endsWith("Rollup")){
      String objectName = method.substring(0, method.length() -
          6)
      Rollup rollup = new Rollup(urlString, objectName, coding)
      return rollup.read(args[0], args[1], args[2], args[3])
    }
    if (args.length == 0){
      return decodedObj.get("$name")
    }
    if (args.length == 1){
      Obj obj = decodedObj.get("$name")
      obj.set(((Obj)args[0]).get())
      Util.write(urlString, decodedObj, coding)
      return
    }
  }
}
```

Usage of the API is similar to the generated API, also providing a type-safe usage but methods not provided by the accessed IoT object obviously can only be detected at runtime. Listing 3.3 shows the usage of the generic API object.

Listing 3.3: Generic API Object

```
def dev = new ObixAPIObj(url,coding,false,null)
System.out.println("actPosSetpExtractAirValue=" + dev.
    actPosSetpExtractAirValue())
dev.actPosSetpExtractAirValue (new Int (100L))
```

3.6 Related Work

Actinium

Apps as Resources

Actinium [14] provides a framework that directly uses the RESTful interfaces of devices to write applications which are realized as scripts running on a container within a central server. Not only the devices offer RESTful interfaces but also the container itself is RESTful. Scripts, called *Actinium apps*, are designed as resources according to the RESTful paradigm. Apps provide their results through GET handlers, can be triggered by POST and can be configured by PUT. Furthermore, the apps can be combined to larger applications via their REST interfaces.

Runtime Container

The container is an application server that allows dynamic installation, updates and removal of scripts. The same app may be required several times, for instance a heating controller has to be instantiated for each room. It is distinguished between installed apps, which is the program code stored under `/installed` and instances created by posting an individual configuration to an installed app. Instances are then stored in `/instances`.

Mashups

The concept that is very successful in the World Wide Web web is to combine several web services to a higher level service. To mash up devices the app has to take a client role. CoAP requests, similar to XMLHttpRequests, are used to communicate with the devices. In addition to CoAP, also HTTP requests are supported to include standard web services in the mashups. The following code snippet taken from [14] shows a sample CoAP request.

```
var req = new CoapRequest();
// request the PIR sensor resource of a mote via CoAP
req.open("GET", "coap://mote1.example.com/sensors/pir",
         false /*synchronous*/);
// with an application/json response
```

```
req.setRequestHeader("Accept", "application/json");
req.send(); // blocking
// and log it to the console after send() returns
app.dump(req.responseText);
```

Implementation

Californium, which is written in Java, is used as CoAP protocol handler implementation. Javascript has been selected as implementation language for the apps due to its widespread user base and the easy support for web service invocation. The *Rhino* Javascript engine, written in Java, is used to run the scripts, the application server itself is a plain Java implementation.

ThingML

ThingML (Thing Modelling Language) is a high level modelling and imperative action language for constrained devices such as micro controllers that intends to replace traditional ways of software development for microcontrollers such as assembly language or C. It is based on concepts of model-driven engineering (MDE) and model driven architecture (MDA). *Things* represent software components or software wrappers of hardware components. They provide a blueprint so that behaviour defined for a thing is defined for all its instantiations. The behaviour of things is defined by a *state machine*, communication is done through *ports* by exchanging *messages* as shown in the following sample taken from [18].

```
thing LedExample includes LedMsgs, TimerMsgs{
      required port Led{
      sends led_toggle
   }
   required port Timer{
     sends timer_set
     sends timer_start
     receives timer_timeout
   }
   statechart LedBlinker init blink{
      state blink{
      on entry Timer!timer_start(500)
         transition -> blink
         event Timer?timer_timeout
         action Led!led_toggle()
      }
   }
}
```

The platform specific model binds the program to the platform it will run on and is called a *configuration* in ThingML. It defines specific hardware I/O for the ports and specific implemen-

tation of the platform functions as shown in the following example for a configuration for an Arduino board.

```
configuration Blink{
   instance app : LedExample
   instance timer : TimerArduino
   connector app.Timer => timer.timer
   group led : LedArduino
   set led.io.digital_output.pin = DigitalPin:PIN_10
   connector app.Led => led.led.Led
}
```

ThingML works as pre-processor generating and being able to mix-in code from C, Java, Scala and other languages.

3.7 Summary

A library implementing a domain specific API for access to oBIX objects handled by an IoT gateway was presented. Using the library it is much simpler to access the IoT objects because the library handles all the communication details and behaves to the client like a local proxy. The proxy contains domain specific methods, either generated from the oBIX contracts or implemented by Groovy meta-programming to provide to the client domain specific and type-safe access. In addition, the API objects provide an encapsulation of the object updating mechanism via observe or the watch functionality so that the client always sees a local, updated proxy object. Clients can handle these proxy objects like standard Java objects. A lobby object provides access to a list of objects and their contracts to allow simple object detection and API object instantiation. Compared with traditional approaches like direct usage of the REST interfaces of the generic SOAP interface this makes usage of the IoT objects much easier and faster.

Bibliography

[1] Internet Protocol, Version 6 (IPv6) Specification. https://tools.ietf.org/html/rfc2460, 1998.

[2] Building Automation and Control Systems (BACS)—Part 2: Hardware. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=29682, 2004.

[3] OBIX Version 1.1 Working Draft 06. https://www.oasis-open.org/committees/download.php/38212/oBIX-1-1-spec-wd06.pdf, 2010.

[4] Constrained Application Protocol (CoAP) draft-ietf-core-coap-12. https://datatracker.ietf.org/doc/draft-ietf-core-coap/, 2012.

[5] Cosm Internet of Things Platform. https://cosm.com, 2012.

[6] The M2MLabs Mainspring Project. http://www.m2mlabs.com, 2012.

[7] BACnet - Official Website of ASHRAE SSPC 135. http://www.bacnet.org/, 2013.

[8] KNX - the worldwide STANDARD for home and building control. http://www.knx.org/, 2013.

[9] OPC Foundation. https://www.opcfoundation.org/, 2013.

[10] G. Aloisio, D. Conte, C. Elefante, G.P. Marra, G. Mastrantonio, and G. Quarta. Globus monitoring and discovery service and sensorml for grid sensor networks. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE '06. 15th IEEE International Workshops on*, pages 201–206, 2006.

[11] C. Bormann, A.P. Castellani, and Z. Shelby. Coap: An application protocol for billions of tiny internet nodes. *Internet Computing, IEEE*, 16(2):62–67, 2012.

[12] M. Jung, C. Reinisch, and W. Kastner. Integrating building automation systems and ipv6 in the internet of things. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pages 683–688, 2012.

[13] W. Kastner, G. Neugschwandtner, S. Soucek, and H.M. Newmann. Communication systems for building automation and control. *Proceedings of the IEEE*, 93(6):1178–1203, 2005.

[14] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *Internet of Things (IOT), 2012 3rd International Conference on the*, pages 135–142, 2012.

[15] Woo Suk Lee and Seung-Ho Hong. Knx; zigbee gateway for home automation. In *Automation Science and Engineering, 2008. CASE 2008. IEEE International Conference on*, pages 750–755, 2008.

[16] Rebeca P. Díaz Redondo, Ana Fernández Vilas, Manuel Ramos Cabrer, and José J. Pazos Arias. Enhancing residential gateways: Osgi service composition. *IEEE Trans. Consumer Electronics*, 53(1):87–95, 2007.

[17] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2010.

[18] Jan Ole Skotterud. High level languages on low level devices : Introducing factorized cross-cutting transitions to ThingML, MA thesis. University of Oslo.

[19] Lorenzo Sommaruga, Tiziana Formilli, and Nicola Rizzo. Domoml: an integrating devices framework for ambient intelligence solutions. In Walter Binder and Heiko Schuldt, editors, *WEWST*, pages 9–15. ACM, 2011.

[20] Eckehard Steinbach, Matthias Kranz, Anas Al-Nuaimi, Stefan Diewald, Andreas Möller, Luis Roalter, and Florian Schweiger. Advances in Media Technology – Internet of Things. Technical report, Institute for Media Technology/Distributed Multimodal Information Processing Group, Technische Universtität München, 80290 München, Germany, January 2013.