“A Device Gateway Service Architecture as a Business Process Integration Enabler in Industry Automation Environments”

Michael Glienecke, Trinity College Dublin (TCD)

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Signed, 20.08.2018

Michael Glienecke

ACKNOWLEDGEMENTS

Writing a PhD thesis is a challenging task which cannot be undertaken alone; it requires many people in various roles and contexts, whom I would like to thank sincerely.

First of all, my deep gratitude goes to my PhD supervisor, Dr. Declan O'Sullivan, for his constant support in all ups and downs and especially his willingness to answer any kind of awkward question without hesitation and absolutely amazing responsiveness for such a busy person. His ability to keep me attached to the thesis and his willingness to accept my more than erratic work schedule due to my professional work in parallel were of paramount importance.

Next, my work colleagues and customers, who shared emotionally and supportive the final stages of the thesis in the last two years with lots of willingness to adjust deadlines, deliverables and provide as much of leeway for writing as possible. Among them would be Dr. Jürgen Hoffmann for his emotional support and especially Heike Rudolph, Derk Kob and Robert Mayrhofer (all from Daimler AG) for providing valuable feedback, discuss and review ideas, visions and critical parts of the technology. With their consent many design ideas and issues could be tested in real-world environments and thus create a much stronger argument.

Finally, my beloved wife Urzula, without her the whole project would never have taken off and who supported me in the bleakest moments when nothing seemed to progress anymore. Her willingness to accept any further direction, even a total stop of the thesis, as well as providing the necessary time for me to write, was paramount in the final achievement of fulfilling the thesis.

SUMMARY

Device gateways as a means of being an abstraction or transparency layer between devices and applications are in use already for a long time in the industry automation area to convert data formats, signals and messages from the physical world of the devices (including PLCs[1] and sensors as data endpoints) to the logical world of the consuming application. However, modern business processes need further value-added services from device gateways to enable an efficient integration of these data endpoints[2]. Devices have to become active participants in the overall business process for enabling the process to adapt to rapidly changing business environments. In addition, semantically enriched data has to be provided, so that consumers can make better decisions based on these. These demands (among others) are cornerstones for the “future of production” [3] as well as part of the larger blueprint of “Industry 4.0”[4].

In this thesis a new architecture for device gateways, which enables the business process integration of devices and sensors as active participants, and the provisioning of additional value-added services and semantically enriched data, is discussed and a reference implementation provided.

After investigation of the current methods of device business process integration (as state-of-the-art analysis) and solutions taken by other device gateway architectures, specific requirements for a new device gateway architecture were defined and discussed in this thesis. Based on these requirements, especially considering the enabling and optimization of the integration of devices into business processes, specific characteristic for the architecture were defined. Based on these requirements and characteristic the design was created and a reference implementation implemented to provide a working solution as well as a test-bed for case studies and experiments.

The evaluation of the design was done with several experiments and two specific case studies, where real-world use cases were implemented using the proposed architecture and Microsoft Azure IoT as an alternative implementation with a comparison and discussion of the results.

---

[1] PLC = Programmable Logic Controller. It is used to control machines in the industry automation area

[2] Abele, Eberhard, und Gunther Reinhart. Zukunft der Produktion. Hanser München, 2011.

[3] ibid

[4] Vogel-Heuser, Birgit, et al. Handbuch Industrie 4.0 2nd Ed Bd.2: Automatisierung (VDI Springer Reference). Springer Vieweg, 2017

Table of content

1 Introduction ... 1

1.1 Motivation ... 1

1.2 Why is this research important? ... 3

1.3 Research question ... 5

1.4 Research approach ... 5

1.5 The reference implementation ... 6

1.6 What is not covered in the thesis ... 6

1.7 Overview of the thesis ... 7

2 Background on Industry automation, production environment, Industry 4.0, integrating devices and their sensors into business processes ... 9

2.1 Devices, sensors, actuators and consumers ... 9

2.2 Industry Automation ... 10

2.2.1 Communication protocols in industry automation ... 10

2.2.2 The typical production (or industry automation) environment ... 11

2.3 Industry 4.0 ... 12

2.3.1 State of practice regarding industry automation and Industry 4.0 ... 12

2.3.2 Industry Automation and adoption rates of IT innovation ... 13

2.3.3 IoT and Industry 4.0 ... 14

2.4 The place of device gateways in automation and Industry 4.0 ... 14

2.4.1 Future trends of device gateways ... 16

2.4.2 Direct integration of devices ... 16

2.5 Real-world integration example in a corrosion quality control lab ... 16

2.5.1 Business process interests and requirements ... 19

2.5.2 Internal requirements ... 20

2.6 State of the art in device gateway architectures ... 21

2.6.1 Overview of existing device gateway architectures ... 21

2.6.2 Gateway architecture feature comparison ... 31

2.6.3 Data protocol support ... 36

2.7 Discussion ... 39

3 Requirements and characteristics of a device gateway architecture ... 41
3.1 Usage scenarios of the architecture ... 41
3.1.1 Corrosion lab scenario simulation ... 42
3.1.2 Exhibition visitor congestion display system scenario simulation ... 44
3.2 Standard use cases to be supported by a gateway architecture ... 46
3.3 Requirements to be supported by a device gateway architecture ... 46
3.3.1 Functional requirements for device gateway architectures ... 47
3.3.2 Non-Functional requirements for device gateway architectures ... 48
3.4 Requirements rating ... 49
3.5 Characteristics of the proposed device gateway architecture - DBGA ... 53
3.5.1 Centralized data store ... 54
3.5.2 Direct data retrieval from data store ... 55
3.5.3 Integrated data quality control ... 55
3.5.4 Semantic data value enrichment ... 56
3.5.5 Communication interface agnostic ... 57
3.5.6 Autonomous operation / workflow execution ... 57
3.5.7 Actuator support ... 58
3.5.8 Write target agnostic ... 59
3.5.9 Data format agnostic ... 60
3.5.10 Preservation of sensor state ... 60
3.5.11 Extendable and easy to change ... 61
3.5.12 Provide feedback on data change ... 61
3.5.13 Basic ontology-Support ... 61
3.6 Comparison of requirements coverage by selected state of art architectures ... 62
3.7 Measuring the improvement achieved by the architecture ... 69
4 Device-Business-Gateway Architecture (DBGA) design ... 75
4.1 Data sources and destinations (data entities) ... 77
4.1.1 Registration of data entities ... 77
4.1.2 Virtual devices / virtual sensors entities ... 78
4.1.3 Data entity trees ... 79

4.1.4 Virtual entity value evaluation (computation) 80
4.1.5 Performance considerations of virtual value evaluation 81
4.2 DBGA communication 81
4.2.1 Communication interface agnosticism 82
4.2.2 Device specific communication 82
4.3 Usage of workflow execution as autonomous operations in the DBGA 83
4.4 The components of the device gateway (logical view) 84
4.5 Process (dynamic) view 85
4.5.1 Device and sensor management 85
4.5.2 Actuator writes 86
4.5.3 Receiver tasks for data endpoint issued writes 87
4.5.4 Sensor scanning task 88
4.5.5 Sensor read 90
4.5.6 Data access 92
4.5.7 Value management 93
4.5.8 Watchdog and cyclic execution with workflows 101
4.5.9 Access control 103
4.6 Physical view 104
4.7 Data model 106
4.8 Discussion 107
5 Reference implementation of DBGA 108
5.1 Options of how to implement the reference implementation 109
5.2 General structure of the reference implementation 111
5.3 CentralServiceLauncher 113
5.4 CentralServerService 114
5.4.1 Modules and components 114
5.4.2 Communication with the outside world 120
5.5 GlobalDataContracts 122
5.6 DeviceServer.Base 123
5.7 DeviceSimulator 123

5.8 GatewayServiceContract ... 123

5.9 Reflection on experience of creating reference implementation ... 124

6 Research Experiments ... 126

6.1 Test equipment used ... 129

6.2 Test #1: Timing to push values into the core with dynamic calculations ... 129

6.3 Test #2: Timing to calculate virtual values in the core using only internal data (available measures) ... 132

6.4 Test #3: Concurrent access (READ / WRITE) by several clients to check for concurrency, race conditions and locking issues ... 135

6.5 Test #4: Test with different data formats and conversion vs. native storage / handling ... 139

6.6 Test #5: Core timing considerations when using workflows (triggering, execution control for long-running tasks) ... 141

6.7 Test #6: Communication mode (REST, WCF using SOAP, .NET Remoting (Binary)) implications ... 144

6.8 Test 7: ODATA access ... 149

6.9 Test #8: Writing actuator values ... 153

6.10 Test #9: MS-MQ adapter to READ / WRITE data with sample consumer / producer to prove the extendibility ... 156

6.11 Test #10: MS-MQ adapter performance to evaluate usage in business workflow environments ... 157

6.12 Overall observations during the experiments ... 159

6.13 Discussion ... 160

7 Case Studies: comparing DBGA and Microsoft Azure IoT ... 163

7.1 Corrosion Lab Scenario ... 164

7.1.1 Device-Business-Gateway (DBGA) based implementation ... 165

7.1.2 Azure IoT based implementation ... 166

7.2 Exhibition visitor congestion display system ... 168

7.2.1 Device-Business-Gateway (DBGA) based implementation ... 169

7.2.2 Azure IoT based implementation ... 170

7.3 Comparison of architectures regarding the case studies ... 171

7.4 Key Findings 176
7.5 Discussion 178
8 Conclusions 180
8.1 Structure of the thesis 180
8.2 Main Findings 182
8.3 Future Work 183
8.4 Contribution 185
9 References 187
A. Appendix 1 - Use Case Details 193
A.1 Corrosion lab 193
A.2 Exhibition visitor congestion display system 198
B. Appendix 2 - Specification Details 201
B.1 Gateway architecture feature definition 201
C. Appendix 3 - Device-Details 207
C.1 Devices considered for the DBGA 207
C.1.1 Physical devices 208
C.1.2 Networked devices 208
C.1.3 Non-networked devices 208
C.1.4 Logical devices 210
C.1.5 Consumers of device data considered 211
C.2 Common operations in device integration 212
C.2.1 Reading device data (including cleansing, filtering and manipulating) 212
C.2.2 Storing, caching and querying of device data 213
C.2.3 Informing consumers about change of data 214
C.2.4 Writing data to actuators 214
C.2.5 Device and operations control (supervision) 215
C.3 Device to business processes integration and integration patterns 215
C.3.1 Point-to-Point integration 219
C.3.2 Hub-and-Spoke integration 221
D. Appendix 4 – Design Details 225

D.1 Data Model ... 225
D.1.1 DbDevice ... 227
D.1.2 DbDeviceAttributes ... 227
D.1.3 DbSensor ... 227
D.1.4 DbSensorAttributes ... 230
D.1.5 DbSensorData ... 230
D.1.6 DbSensorDependency ... 230
D.1.7 TrackingPoint ... 230
D.1.8 Log ... 230
D.2 Data consumers for the device gateway ... 231
D.3 DBGA communication handler details ... 232
D.4 Process (Dynamic) view details ... 235
D.4.1 Receiver tasks for data endpoint issued writes ... 235
D.4.2 Callback handling ... 237
D.4.2.1 Callback example in Python ... 239
D.4.3 Error- and Log handling component ... 239
D.4.4 Gateway engine core ... 240
D.4.5 Dynamic configuration ... 240

Table of Figures and Tables

Figure 2.2.1: Device Gateway for the Internet (IIoT gateway) 15
Figure 2.2: Current Corrosion Lab Process 19
Figure 2.3: Xively services 23
Figure 2.4: Azure IoT solution architecture [81] 25
Figure 2.5: MyDriving sample application in Azure IoT showing sub-services used 26
Table 2.1: List of features 32
Table 2.2: Feature overview 33
Table 2.3: Data protocol support 38
Figure 3.1: UML Use Case diagram for Corrosion Lab 43
Table 3.1: Specialized use cases for Corrosion Lab 43
Figure 3.2: UML Use Case diagram for exhibition visitor congestion display system 45
Table 3.2: Specialized use cases for exhibition visitor congestion display system 45
Table 3.3: Functional requirements for device gateway architectures 47
Table 3.4: Non-Functional requirements for device gateway architectures 48
Table 3.5: Requirements/Characteristics rating for Industry Automation/Business Process integration 50
Table 3.6: Table of characteristics 53
Table 3.7: Mapping between characteristic and requirements 54
Table 3.8: Requirement coverage for device gateway architectures including DBGA 63
Table 3.9: Measurement criteria definition to compare different architectures 70
Figure 4.1: Data Entity Tree 80
Table 4.1: Use-Case to Component Mapping 84
Figure 4.2: UML Component Interaction 85
Figure 4.3: Writing actuator value 87
Figure 4.4: Receiver tasks for endpoint write requests 88
Figure 4.5: Task Launcher - Scanner Task - Sensor 89
Figure 4.6: Operations inside scanner task 90
Figure 4.7: Retrieve values as a consumer 92
Figure 4.8: Data access class diagram 93
Figure 4.9: Current Sensor Value Management (read / write) 96
Table 4.2: Callback type and use 98
Figure 4.10: Virtual Value Evaluation 100
Figure 4.11: Cyclic Task Execution 102
Figure 4.12: Access control 104
Figure 4.13: Exemplary physical view of device gateway 106
Figure 5.1: Reference Implementation (Main) Components 113

Table 6.1: Experiment overview ... 127
Table 6.2: Test 1 results in μsec for 10,000 executions ... 131
Figure 6.1: Test 1 results in μsec for 10,000 executions ... 131
Table 6.3: Test 2 results in μsec for 10,000 executions ... 134
Figure 6.2: Test 2 results in μsec for 10,000 executions ... 134
Table 6.4: Test 3 results in μsec for 5 parallel read and 5 parallel write executions (500 times each) ... 137
Figure 6.3: Test 3 results in μsec for 5 parallel read and 5 parallel write executions (500 times each) – reading operations ... 137
Figure 6.4: Test 3 results in μsec for 5 parallel read and 5 parallel write executions (500 times each) – writing operations ... 138
Table 6.5: Data format conversion performance of SQL, Python and C# using native and agnostic data formats (in μsec) ... 140
Table 6.6: Data format conversion performance of SQL, Python and C# using native and agnostic data formats (in μsec) ... 140
Table 6.7: Workflow as trigger for checking and after change (in μsec) ... 143
Table 6.8: Workflow as virtual value calculation (in μsec) ... 143
Table 6.9: Workflow in a parallel read / write scenario (in μsec) ... 143
Table 6.10: 1, 5 and 10 threads in parallel retrieving data using .NET Remoting, SOAP and REST as protocol (in μsec) ... 145
Table 6.11: 1, 5 and 10 threads in parallel retrieving data using .NET Remoting, SOAP and REST as protocol between 2 machines (in μsec) ... 146
Figure 6.5: 1, 5 and 10 threads in parallel retrieving data using .NET Remoting as protocol (in μsec) ... 146
Figure 6.6: 1, 5 and 10 threads in parallel retrieving data using SOAP as protocol (in μsec) 147
Figure 6.7: 1, 5 and 10 threads in parallel retrieving data using REST as protocol (in μsec) 147
Figure 6.8: ODATA server processing time in msec for 100 - 9,900 records ... 151
Figure 6.9: ODATA client request time in msec for 100 - 9,900 records ... 151
Figure 6.10: ODATA server processing time in msec for 5,000- 195,000 records ... 152
Figure 6.11: ODATA client request time in msec for 5,000- 195,000 records ... 152
Figure 6.12: total time in μsec for writing 1 sensor value to an actuator ... 154
Figure 6.13: total time in μsec for writing 20 sensor values to an actuator ... 155
Figure 6.14: time in msec for sending a request including receiving response using MS-MQ from client to server ... 158
Figure 6.15: time in msec for receiving a request, processing and posting an actuator request and response packet using MS-MQ on the server ... 159
Table 6.12: Test result overview ... 161

Figure 7.1: Schematic overview Corrosion Lab scenario using DBGA 166
Figure 7.2: Schematic overview Corrosion Lab scenario using Azure IoT 167
Figure 7.3: Exhibition Visitor Scenario implementation using DBGA 170
Figure 7.4: Exhibition Visitor Scenario implementation using Azure IoT 171
Table 7.1: Comparison of different architectures 172
Table 8.1: Future work overview 184
Table A.1: Corrosion lab scenario complete use case table 193
Table A.2: Exhibition visitor congestion display system scenario complete use case table ... 198
Table B.1: Gateway architecture feature definition 201
Figure C.1: Logical Device Chain 211
Figure C.2: QA process 217
Figure C.3: Point-to-Point integration 220
Figure C.4: Hub-and-Spoke integration 222
Figure D.1: Data Model 225
Table D.1: Sensor definition attributes 227
Figure D.2: Communication interfaces 233
Figure D.3: Callback types 237
Table D.2: Callback kinds 238

1 Introduction

1.1 Motivation

Device gateways, as a means of being an abstraction or transparency layer between devices and applications that use them, have been around since the late 1980s primarily in industry automation environments (usually production) [84]. Their main purpose was to shield the consuming (using) application from the intricate details of the device inter-operation, such as protocols used, hardware-specific issues like byte-ordering, and so on. These early gateways existed mostly to make application development easier, as either dedicated hardware systems or libraries / linkable modules.

In the early 2000s, as companies started to try to bring the classical business process closer to the shop floor and its automation systems for enhanced control and optimization, more generalized software like TinyDB (2003) [79], IrisNet (2002) [80] or OPC-CA [86] were introduced. This was manifested as independent middleware between devices and consumers, in the wake of service orientation in general. From this point onwards, the device gateway was considered an independent service which sat in the middle between devices and consumers to handle all issues. In addition to the physical handling of protocols and hardware, the addressing of resources was a major concern as well.

In the wake of the discussions around REST[5] vs WS-* based solutions (starting from the early 2000s) with projects like SIRENA ([88], [89]) and SOCRADES [90], accessing devices using URL based schemes (REST) also came to the fore. This was the approach taken by device gateways like SenseWeb [7] and various others to hide the device behind http-requests using interfaces like [2], [3] and [5]. In addition, service bus architectures have been developed [9], [81] (Microsoft Azure IoT being one example), but these are used more in environments where the direct interaction with the device is not so much an issue.

So, in summary a device gateway is currently generally considered to be a more or less a simple façade (with different addressing options) for devices, but usually do not provide additional value or services to aid business processes and workflow integration.

However, in the business world, more and more businesses need to apply formal business processes, in order to be able to quickly adapt to new requirements more easily [62]. Aside from the adoption of formal business processes, comes the need to undertake additional

---

5 REST = Representational State Transfer
(http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)

control and performance measurements, so as to support a more objective view of the real business performance and its parameters. Together with other needs like consumer-customized products at mass production prices, leaner production to save resources and enhanced value-chains, leads to concepts like "Industry 4.0"[6] [84] which drives the next generation of device gateways.

As acknowledged widely, e.g. [25], [87], it is clear that devices must be part of integrated business processes and thus value-chains, in order to be able to achieve the desired needs of business as discussed above, which in a nutshell is the ability to enable rapid change whilst being able to monitor and control performance and generate additional value.

Newly emerging business models in the industry automation-world utilize device integration to a very high degree and are explicitly dependent on it to function [60]. In [84] this is clearly stated: "for producing companies there is a clear demand to integrate what happens on the shop floor with the global IT-systems".

The business process integration of devices could be either done using a direct device integration[7] (for example reading the consumer entrance counter of a gate, combining it with the current time and adjusting the price of the merchandise in a shop – more in section 2.4.2), or as semantically enriched (added value) data integrated into the process [5], like in historical averages which can be easily consumed, lookups performed based on other data, and so on. This data enrichment would usually be the task of a device-gateway, yet as such gateways (as will be shown later) do not offer these features easily there is a need for a new architecture to provide exactly this.

In summary, there is a driving need for a device-gateway architecture that fully integrates devices as cyber-physical entities into business processes in order to make them *active participants* in the process, in addition to providing added data (as new business intelligence) to the process, therefore improving the global value-chain. Thus, in addition to the classical device-gateway features that one would expect, the new device-gateway architecture would need to support:

- services for business process and workflow integration of devices;

---

[6] Industry 4.0 is the complete integration of the production environment into an overall IT-landscape to provide "smart factories". These "smart factories" are the underlying foundation for the challenges and requirements of the 21st century regarding industrial production (https://www.bmwi.de/Redaktion/DE/Dossier/industrie-40.html)

[7] Direct Device Integration is when a device is directly integrated into any kind of further process without an intermediary of any kind [84]

- provision of added value to business processes by means of semantic enrichment[8] of data;

This type of device gateway architecture - the joining and merging of device and business world requirements - has had little research focus, prior to the research described in this thesis, and it is argued that the outcomes of the research presented in this thesis is unique and a contribution to both research and business communities.

1.2 Why is this research important?

The core device integration using protocol adaption and conversion has been researched for a long time as has been storage structures, communication patterns with devices and consumers (protocols in general). Therefore, these areas are considered as being more or less defined and it is considered that very few new contributions can be achieved here.

Yet, as foretold in the "Strategic Research Agenda" of EPoSS[9] ([29]), smart systems are emerging in all domains, and the data integration, processing and further usage of sensing data in processes are of paramount importance in these domains (for example medical systems, automotive, Internet of Things).

Given the trends, and the requirements and challenges outlined in section 1.1, it was very clear from the outset of the PhD that research was (and continues to be) needed in this area. Especially issues as how data can be semantically enriched and thus made "more valuable", fed into business processes, as well as taking part in business process interactions are very important. Another issue, when looking at "smart systems" is the question of how they are becoming "smart". Here workflows, which allow them to be smart and self-controlling / resilient, play a major role. With the growing importance of "Big Data"[10] in the industry [93], [87] the provisioning of highly aggregated and pre-computed data from industry automation

---

[8] The term "Semantic Enrichment" is used throughout the thesis in a sense that additional data is made available for existing data by means of giving the existing data additional meaning, usability and value. This is achieved by adding contextual data to the existing data (so it's semantic usefulness increases – it becomes more meaningful for consumers)

[9] EPoSS is an industry-driven policy initiative, defining R&D and innovation needs as well as policy requirements related to Smart Systems Integration and integrated Micro- and Nanosystems. EPoSS is contributing to EUROPE 2020, the EU's growth strategy for the coming decade, to become a smart, sustainable and inclusive economy (http://www.smart-systems-integration.org/public)

[10] Big data is an evolving term that describes any voluminous amount of structured, semistructured and unstructured data that has the potential to be mined for information (https://searchdatamanagement.techtarget.com/definition/big-data)

also becomes a necessity and thus device gateways as places where this aggregation and calculation can happen, become more important for research. In places where this is used already (like the evaluation and definition of the stress collectives on parts as indicated by [94]) it clearly shows that these highly aggregated data measures (e.g. the "quality of a measurement") become important decision factors in the overall part approval business process and therefore the whole supply chain management.

Another reason as to why the research is important is the fact that there appears to be a tendency to connect devices directly to the business process (and therefore consumers) using direct device integration approaches (see 2.4.1). This might provide quick-wins for the involved parties as results are easily obtainable (and rather cheap as well) with a certain feeling of "agility". Yet in the long run every system which evolves over time will face the old dilemma that maintenance and support gets harder and will cost more, whereas extendibility and innovation will decrease. The author of this thesis (based on extensive experience in the business process integration domain in automation) personally considers this direct connection approach very problematic due to the following reasons:

- Security control would be a task of each device and not a task of a centralized, more superior instance (like a gateway);
  A classic example would be that for instance Siemens S7 PLCs[11] which has no firewall or other protection against malicious communication and writing to an output area could result in disastrous machine behavior.
- Systems become quickly very complex and very fragile (devices not responding, timing issues, reliability of information);
- A controlling instance in the "middle" seems important as otherwise the old problem of n..m relations (consumers – producers) arises;
- In general, it is often underestimated how much effort has to be put in even the simple task to coordinate a small number of devices all speaking the same "language" (for example REST). Still every device is different, has different semantic requirements and responds in different ways.

It should be noted that the idea to publish devices on the Internet by means of an intermediary like a gateway is in the opinion and experience of the author of this thesis a very good and viable idea; with no direct access to the device, but access always controlled and

[11] PLC = Programmable Logic Controller – a device to control a machine in a guaranteed way usually used in industry automation environments

within operational limits. Here the device gateway could act as a much better suited "guardian" for the information, privacy and communication issues involved.

In this way a device, or even a group of devices – this abstraction could be taken higher and higher (up to for example a complete corrosion test unit) – could be represented as a "device" or even more a "service" which can be integrated into business processes.

Many of the issues discussed above are similar to IoT middleware systems where a lot of research has been undertaken [91]. Yet as current IoT middleware is typically wireless sensor network (WSN) centric, many of the implications and assumptions do not translate well to the automation and production environment. For instance in automation/production environments the availability, adequacy and stability of power supply, computing power and communications are usually a given. In any case it is explicitly acknowledged that no single IoT middleware can support all environments and requirements [91].

1.3 Research question

Given the motivations outlined in sections 1.1 and 1.2, ***this research examines the extent to which a new device gateway architecture (with integrated data quality control, increase of data value by semantic enrichment, communication interface agnosticism, data target independence and data format agnosticism) that is available for low cost will improve business process integration in environments where industry automation is used***.

1.4 Research approach

To be able to test the various assumptions, requirements and characteristics of the proposed gateway architecture (from now on called the **'Device-Business-Gateway Architecture (DBGA)**'), a complete reference implementation was created, as well as an examination of existing state of the art solutions and architectures. Experiments based on the reference implementation were undertaken in various machine constellations to:

1) Evaluate the overall stability, reliability and performance, as well as timing and load characteristic for several use cases;
2) Perform functional tests of compliance with design principles (like correct evaluation of virtual values or proper filter of invalid values);
3) Provide as close as possible to a real use case scenario, so that a later implementation in real world projects would not have any hidden surprises.

Furthermore, the proposed gateway architecture has been evaluated in two case studies. Both scenarios (one from a customer of the author's, one from an "art exhibition" domain) are implemented using the architecture and compared to Microsoft Azure as an alternative approach.

In general, the research is using a very practical approach and focused on real-world scenarios compared to a more purely theoretical and comparative approach.

1.5 The reference implementation

The reference implementation, which is available as open source[12], was implemented using C# in the .NET 4.6.1 environment using available free of charge products (like SQL Server 2012 Express as a database and Visual Studio 2015 Express Edition), and therefore all parts of the solution can be run without using royalty-based software (except Windows as such – here the .NET Platform on Linux or Mac[13] might be an alternative). On the connected physical clients, the .NET Micro Framework 4.4 (some devices required 4.3) was used[14]. For Python and Ruby, the corresponding packages IronPython 2.7 and IronRuby 1.1.3 were used. JSON[15] was implemented using the fastJSON 1.0 library.

As a handler for the WS* and REST-requests the WCF 4.5 (Windows Communication Foundation) infrastructure in IIS 7 was utilized to have an easy, declaration based (contract first) development environment, which is much easier to maintain and extend. Should the need arise this can be changed easily to a more classical request-handler-loop approach.

All implementation specific parts can be exchanged as dynamic handlers are used which will for example allow the exchange of SQL Server with mySQL, and so on. As a sample for this technology a pluggable data store to XML files has been provided as well.

1.6 What is not covered in the thesis

The Internet of Things (IoT) at the moment is the main driving force behind the motivation of many companies, research institutions and people to use, research, manage and integrate devices. In addition, many new devices and usage scenarios are being developed with this focus. Therefore, it is quite natural that many questions and thus research issues arise from this environment. Combining with the implications and requirements of Industry 4.0[16], being the 4th industrial revolution, will also result in even more complex questions arising. In particular, the combination of industry automation, sensor systems, business processes and

---

[12] https://github.com/mglienecke/DeviceGateway

[13] https://channel9.msdn.com/events/Build/2015/3-670 and https://github.com/dotnet/core

[14] https://github.com/NETMF/netmf-interpreter/releases

[15] JSON (JavaScript Object Notation) is a lightweight data-interchange format http://www.json.org/

[16] Industry 4.0 – the next industrial revolution (https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html)

knowledge management are of paramount importance for the successful enterprise of the 21st century.

This thesis however only focuses in particular on the device integration part – the part where the physical device meets the business process.

This thesis **does not cover** wider IoT issues, nor does it cover industry automation in general, nor sensors as such, except where needed for clarity.

All these topics are part of a wider discussion, and this thesis can be considered an important cornerstone in the overall solution, as sensors and the gateway to them, is the basis for any further data analysis, process integration and knowledge gain.

In addition, this thesis briefly considers mobile devices like smart phones, and so on, mainly for their integrated sensing abilities and sensors, and not for their classical use model (which would be a desktop or laptop computer replacement, used and operated with the same interfaces like web browsers and so on). Thus, for this thesis, mobile devices are only relevant because they possess quite considerable calculation power, memory, storage, sensors and especially user interface in one unit – so they can manage the sensors they contain, communicate with a gateway and a business process (via the gateway) and interact with the user.

1.7 Overview of the thesis

After the introduction, the thesis in chapter 2 provides background information about business processes, integration patterns, and key issues regarding device integration (devices, consumers, operation modes). In addition, the chapter presents a state of the art overview of the currently available technologies and implementation patterns for doing business process integration.

Chapter 3 presents the key requirements as well as the main characteristics for the proposed DBGA, which were derived from the state of the art and also presents two use cases. These use cases will be further used to evaluate the DBGA (and reference implementation).

The DBGA is described in detail in chapter 3.7 and full details about the reference implementation of the DBGA is given in chapter 5.

Chapter 6 describes several experiments which were undertaken to evaluate the architecture. This is further exemplified by chapter 7 where two case studies (defined in section 3.1) and their possible implementation using the DBGA and one existing state of the art architecture (Azure IoT) are discussed.

Final conclusions are drawn in chapter 8.

Additional information towards the presented topics is provided as appendixes in appendix A, B, C and D.

2 Background on Industry automation, production environment, Industry 4.0, integrating devices and their sensors into business processes

This chapter provides background information about industry automation, Industry 4.0, and devices and definitions covered (section 2.1, 2.2 and 2.3). Following on, there are sections about the place device gateways have in industry automation (2.4), a real-world example how a device-gateway can be utilized to achieve business process integration (section 2.5) and a review of the state of the art in Gateway Architectures including a comparison of their features (section 2.6) followed by a discussion (section 2.7).

2.1 Devices, sensors, actuators and consumers

For any device gateway, devices, sensors, actuators and consumers are the main players and therefore it is important to properly define the terms as well as the usage of them (for details of devices, typical operations and integration patterns see appendix C).

In this thesis the following definitions for terms are used:

- A **sensor** responds to a physical stimulus (as heat, light, sound, pressure, magnetism, or a particular motion) and transmits a resulting impulse (as for measurement or operating a control)[17].
- An **actuator** moves or controls something[18].
- A **device** is a piece of equipment (e.g. a machine) or a mechanism designed to serve a special purpose or perform a special function[19] with sensors and / or actuators attached to it.

Very often in the corresponding literature (and in this thesis as well) the term device and sensor / actuator are used interchangeably[20] which indicates that a device is operating as a sensor / actuator (which can be seen by the above definitions as well).

This can be considered true, if either the device has exactly one sensor / actuator and therefore 1 device equals 1 sensor (or actuator), or a device is referenced in a general term – thus regardless of the number of sensors / actuators involved. Internally, for the Device-Business-Gateway Architecture (DBGA), device and sensor / actuator are separate data definitions and entities. A device always contains 0..n sensors / actuators.

---

[17] http://www.merriam-webster.com/dictionary/sensor

[18] http://www.merriam-webster.com/dictionary/actuator

[19] http://www.merriam-webster.com/dictionary/device

[20] Like: “the device sends” or “data is written to the device” instead of “sensor X send” or “data is written to actuator Y”

In general sensors, are read-only and actuators are write-only, with some exceptions where for instance a sensor can be written to, or an actuator read from, as well.

2.2 Industry Automation

Since the late 1960s industry automation was defined as: “automatically controlled operation of an apparatus, process, or system by mechanical or electronic devices that take the place of human labor”[21].

This controlled operation involved mostly two components:

- The machine, which is to be controlled
- Programmable Logic Controllers (PLCs), as very resilient autonomously runnable building blocks to actually control the machine operation

Especially the PLCs are and were of paramount importance in the industry automation environment as they guarantee that timing considerations from the physical process are met, security issues considered (e.g. protect the human operator with security checks) and overall that the machine as one unit is operational. Over the past decade the PLC market[22] is quite segmented with Siemens as the clear leader in Europe.

2.2.1 Communication protocols in industry automation

Industry automation – due to the nature of the physical process and the machine control – requires un-interrupted, guaranteed, reliable and deterministic communication between participants.

To achieve this, many protocols have evolved over time starting with serial protocols (e.g. R3964) towards the now network oriented ones. Yet even nowadays, sometimes due to hazard prevention (e.g. handling of explosive materials), sometimes due to machine investment cycles (which can be 15 – 20 years), serial communication can still be found in many places.

The networked environments can further be divided[23] into the so called “field-bus”-systems[24] like ProfiBus (Siemens), DeviceNet (Rockwell), CANOpen, etc. , and Ethernet-systems like Modbus-TCP, EtherCAT, and so on.

---

[21] https://www.merriam-webster.com/dictionary/automation

[22] Overview of Top PLC manufacturers: http://automationprimer.com/2013/10/06/plc-manufacturer-rankings/

[23] http://www.feldbusse.de/#feldbusse-uebersicht

[24] Overview of field-bus-systems: http://www.feldbusse.de/Vergleich/uebersicht_feldbusse.shtml

These Ethernet-systems share many similarities with the classical office-Ethernet-infrastructures, yet features like guaranteed response times, real-time ability with update cycles of <= 1ms were missing and had to be added. To give an example, with EtherCAT the update time for 1000 I/Os is only 30 µs including I/O-time. With a single Ethernet-frame (which can be transferred in 300 µs) up to 1,486 byte of process data can be exchanged which equates to nearly 12,000 digital I/O ports.

2.2.2 The typical production (or industry automation) environment

In a typical production environment, the "shop floor" is where all physical operations (the production) is handled. Most PLCs and machines would be located and operated there. Logically on top of the shop-floor is the shop-floor management[25] which ensures the smooth overall operation of one or several shop floors. Usually the shop-floor management uses SCADA-Systems[26] to visualize and control operations of several sub-systems (machines). In parallel most companies would use ERP/MES-systems[27] to integrate the process data into further business processes and areas like product lifecycle management, customer relationship management, and so on.

Parallel to the core production environment are usually other environments, which include machine control as well, yet are not directly involved in production, but associated with it.

A typical example (which will used as a running example in this chapter) would be the corrosion test lab of Daimler AG which is associated with the surface production unit and department, and yet has no direct role in production.

As the production environment, being the core revenue generating process, is shielded and protected, and thus usually not (easily) accessible from the outside, systems in parallel zones are often having the same (or slightly lesser) restrictions.

This means that in the above-mentioned example the climate chamber – as an independent "machine" with a physical control and a PLC to control it – cannot be directly accessed from outside of the local environment, but needs interfacing using an intermediary.

As these secondary machines (or devices in general) are not production-relevant areas, they are not integrated into any SCADA, ERP- or MES-lifecycle and usually left alone as non-integrated information islands.

---

[25] Definition of Shop-Floor Management: http://www.refa.de/lexikon/shopfloor-management

[26] SCADA = Supervisory Control and Data Acquisition (https://en.wikipedia.org/wiki/SCADA)

[27] ERP = Enterprise Resource Planning, MES = Manufacturing Execution Systems

2.3 Industry 4.0

The goal of "Industry 4.0" as the 4th Industrial Revolution (after steam-engine, conveyor-belt and computer) is in principal the complete integration of the production environment into an overall IT-landscape to provide "smart factories"[28]. These "smart factories" are the underlying foundation for the challenges and requirements of the 21st century regarding industrial production ([95]).

The aim is to create dynamic value-chains which include all phases and cycles of a product from design and development over customization to production and delivery. These value-chains can then be optimized (and combined and integrated) to provide better profitability, faster reaction and production times, or in general a better utilization of resources like raw materials, supply-chain-management and finally customer satisfaction and retention.

A basic necessity for this to function is the complete networking and integration of all parts and components along the value-chain, including secondary systems, which might be relevant in terms of quality assurance, and so on. In our Daimler climate chamber example, it is for instance mandatory that sample data which points to supply problems (e.g. raw material corrosion) is immediately fed back into the supply-chain and production life-cycle to prevent further damage much later (even years after production).

Using such an approach it will be possible – at least this is the goal – to produce highly individual goods with excellent support for humans in the production process [103], with the production methods (and price-levels) of mass-production, streamlining logistics and supply chains as well ([100]). In the value-chain every step is then – by using the necessary automation and integration – streamlined and configured to achieve this.

2.3.1 State of practice regarding industry automation and Industry 4.0

Industry automation in the Western Industry Nations is quite common and normal as can be seen for example in the usage of industry robots per 10,000 workers. Whereas the global average is at 74, many of the leading industry nations far exceed that level (with South Korea at 631, Germany at 309 or Switzerland with 128).[29] On the other hand, as Industry 4.0 is still quite new (projects mainly started in 2015 / 2016), the establishment of the mechanisms in factories is, if already started at all, an ongoing process. Currently – according to the German Ministry of Economy – 20% of all German industry producers are already utilizing self-controlling processes and 83% are considering a high digitalization of their value chains until 2020, with planned investments into Industry 4.0 of 40 billion EUR per year until 2020

---

[28] https://www.bmwi.de/Redaktion/DE/Dossier/industrie-40.html

[29] https://blogs-images.forbes.com/niallmccarthy/files/2018/04/20180425_Robot_Workers.jpg

followed by further 33 billion EUR for additional investments in IT, electronics and machines from 2016 – 2030 [107]

So generally speaking, most modern factories have already established basic industry automation, yet the aim of a digital value-chain is quite far away for most. Especially when it comes to secondary (not directly production relevant) systems this is even more the case.

2.3.2 Industry Automation and adoption rates of IT innovation

Industry automation utilizes Information Technology (IT) concepts, methods and processes to achieve its goals. Yet the classical IT infrastructure of an industry automation supported company is normally far more focussed on terms like payroll / order processing, customer relationship management, computer management, and so on, rather than the industrial application/business process itself.

Whereas in the technical world of industry automation very "real" factors like response time in msec, size of a telegram in bytes and even bit-alignments matter, these are very seldom issues for classical IT-departments. In addition, due to the importance of the production and the longevity of investments in that area[30] industry automation usually follows different timescales and development cycles as well[31]. Another factor is that failures of a system can easily cause harm to humans (and other systems) whereas in "normal" IT the worst-case scenario usually is data- or money-loss[32].

All this leads to the perceived impression that development and innovation adoption progress is not as fast in industry automation than in "normal" IT – mainly due to the above-mentioned innovation cycles and restrictions.

This has to be taken into account when developing architectures and solutions for this area. It implies that for instance new technologies are only introduced in the industry automation domain after maturity of a technology is assured. As an example, Microsoft .NET (which was generally available since 2002) has been provided as Microsoft .NET Micro Framework for use in industry automation environments from 2009 onwards. Another example would be SOAP, which was available roughly around 1999 – but usage in the industry automation

---

[30] In machines investments cycles of 10 – 15 years are not uncommon whereas in IT usually 4-5 years are the maximum (which can be seen in depreciation tables for taxation as well)

[31] Which is exaggerated by the fact that usually after a machine is operational there is no real test-environment to check new features aside taking the machine out of production

[32] Exceptions to this include among others medicine, aviation, military operations, etc. Yet here the same time scales of innovation adaption can be found.

device world did not really take place before 2007 with the advent of "device profiles for web services".

The implication is that usually adoption of new IT technology into industry automation environment is typically delayed between 5 – 10 years after adoption into "normal" IT environments, and the technology has to undergo a much more stringent check and usually "hardening" phase where as many as possible problems are eliminated, before it will be utilized in the industrial automation world.

2.3.3 IoT and Industry 4.0

Industry 4.0 and the Internet of Things (IoT) are often similar, yet they are still different approaches.

In IoT the main focus is on integration and mass connectivity of thousands (or millions) of sensors, quite seldom actuators, very often associated with consumers to create new intelligent services based on the generated data. This comes with all associated problems like device energy consumption, connectivity problems, downtime, ability to process the data in general, and so on.

Industry 4.0 (sometimes called the industrial internet as well) on the other hand considers industry with stable energy supply, networking infrastructure, etc. – where the focus is more towards the value-chain and the change involved there.

Yet both share aspects like Internet connectivity between value-chain partners (e.g. supplier and producer), cloud-based computing and as a long-term goal a unified world-view where the physical and virtual world come together to form one common system - see [84], [102] and [96].

2.4 The place of device gateways in automation and Industry 4.0

Every machine (or device in general) which is supposed to be integrated into a business process or value-chain has to be accessed via a communication protocol. In some cases, this is done from a consumer via direct access to the device (usually a PLC) using the native protocols (e.g. Profibus), yet in most cases an intermediary – sitting between consumer and producer (the device or machine) – is used. This intermediary, the device gateway, acts as an access point to the device.

In the past device integration was – if it was performed at all – mostly only an integration suited for the purpose it was intended for, e.g. a SCADA-system which means that the data could not be used by anybody else as it was a 1:1 integration.

Due to the open information flow envisaged in Industry 4.0 – especially towards the cloud as participants of the value-chain are geographically dispersed – the gateway becomes more of an Internet to Device (or IIoT) gateway[33], see Figure 2.1.

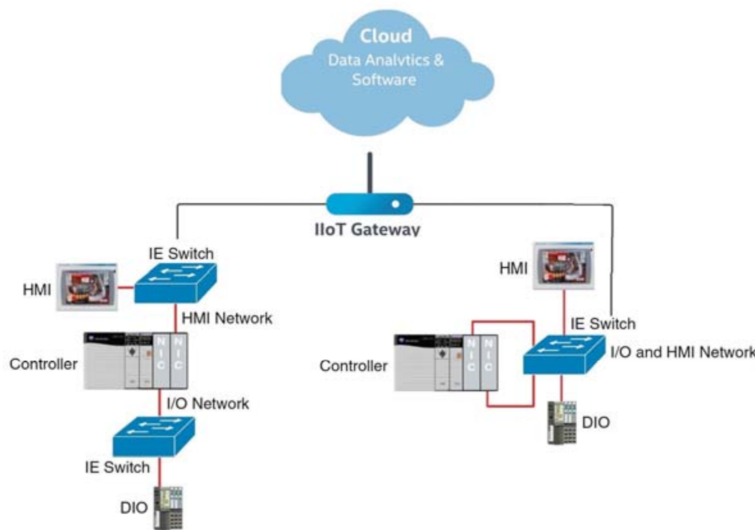

Figure 2.2.1: Device Gateway for the Internet (IIoT gateway)[34]

In addition to being "just" a gateway, due to its computing abilities, the gateway very often takes the role of an edge device in a fog computing environment[35]. Operated in such a mode, the gateway becomes what is known as "Soft PLC" [36] – a PLC running on a standard system and not on dedicated hardware.

Especially with the advent of business-process and value-chain integration, device-gateways become more important and "first-class citizens" in the industry IT landscape. This point is neatly encapsulated by the quote: "in the context of ever-increased networking of devices, the amount and heterogeneity of interfaces to be considered increases […] which creates a demand to simplify the integration and programmability of these systems, so that they are usable in changing environments for changing requirements" [84].

---

[33] Acronym not to be confused with IoT (internet of things), which is entirely different

[34] https://industrial-iot.com/2016/07/future-industrial-gateway/

[35] Fog Computing is an architecture where edge devices perform physical input and output and carry out a substantial amount of computation and storage (https://en.wikipedia.org/wiki/Fog_computing)

[36] A Soft-PLC is a PLC running only in software – often as a service. This requires a very stable infrastructure as the same requirements towards resilience, fault-tolerance and timing apply very often like towards regular PLCs.

2.4.1 Future trends of device gateways

As envisioned by [84], [87] and [97] as well as by this thesis author's own experience on working upon leading edge customers' projects, device gateways will develop very much like shown in Figure 2.2.1. They will become hubs to gather data and represent often autonomous intelligence ("Soft-PLC"). In addition, the generated data is used in further data analysis (Big Data) by feeding it usually into so called data-lakes [105].

Here not only the raw values, but especially the virtual sensor values (see section 4.1.2) are of paramount importance, as they represent highly aggregated value[37] which is created close to the origin with lots of expertise which has not to be replicated at levels much further away[38]. These generated values can then be used "as is" and / or processed further to form new information levels.

2.4.2 Direct integration of devices

In contrast to an integration using device gateways, as described in 2.4, very often (according to the author's experience) devices are integrated directly on a 1:1 basis with the consuming process. Thus, without the intermediary, the device gateway, which is why this method is termed "direct integration" by the author and some references (like in [84])

In general, this approach offers quick results and usually fast integration times, whereas (based on experience) in the long run the total costs and effort needed to maintain a properly working solution far outweighs any short-term gain.

This in addition to other, already mentioned criteria like increasing complexity, security issues and coordination renders this approach, in the opinion of the author, is not suitable for larger integration scenarios, or when issues like data quality, value generation for the overall process including data retrieval, etc. arise.

However, it is a good option to consider for simple integration of just some device data (usually read-only).

2.5 Real-world integration example in a corrosion quality control lab

To give a real-world example for business process integration of a device gateway architecture, a case study from the author's current work, is presented. Daimler AG (the mother company of Mercedes-Benz) has a lab for corrosion tests in Sindelfingen (Germany)

[37] Often these are KPI (Key Process Indicators) or KBI (Key Business Indicators)

[38] "local" knowledge is usually more technology oriented and up-to-date than further away knowledge which has to be taught as well

where around 25 climate control chambers are running different test sets for corrosion checks. These tests involve things like:

- Tempering parts to 70°C and then cooling them to -40°C while being sprayed with hot brine (up to 5% salt in water);
- Permanent heat tests;
- Stone hit tests;
- Scratch tests;
- Lamination tests;
- Deformation tests.

In general, any part (from a tiny screw to a complete van chassis), being used in any produced vehicle by Daimler AG, will have to undergo a test prescribed by internal specifications which define how much corrosion, scratch, delamination, and so on is allowed for a given material with a given treatment, coating, and so on.

So, the typical corrosion test process would be:

- Receive the part and label it with a water-resistant RFID tag which acts as a print label at the same time;
- Place the part into a test chamber;
- Run the tests;
- If a chamber needs cleaning, maintenance, change, the part will be taken out and the tests continue in another chamber;
- Finalize the results;
- Create the reports.

These climate chambers run software (firmware and communication protocols) from the period 1977 – 2018 using 5 different protocols. These range from 2 very rare ones (using RS-232 communication with 300 Baud and MODBUS (J-BUS)), to a Siemens S7 PLC as chamber control, a REST (based on http) to a web-enabled device and proprietary TCP/IP protocols. So, a classic case of networked and non-networked integration.

Each climate chamber usually has sensors for humidity and temperature with optional additional sensors for in- / outflow of brine, brine temperature, gas pressure, content of Cu (copper), Na (sodium), Cl (chlorine) in water. Very modern chambers provide sensors in a redundant configuration so that always the most likely value is returned (determined by the chamber control system). So in total a chamber provides from 2 to 10 sensor values.

Sensor data is read every 10 seconds, thus 6 samples per minute per channel, thus causing a load of 300 to 1500 bytes per second per chamber (including the protocol overhead). In

addition to the chamber data the RFID tags cause the RFID infrastructure to generate RFID movement messages per part so each part can be localized and its exact position over time is recorded.

Data integrity and quality used[39]:

- Data is delivered in different scientific units as some channels report in Celsius, some in Kelvin, some in mBar, some in hPa, and so on.
- Data representation uses different floating-point formats and different bit ordering schemes;
- Data storage is undertaken in 6 different ways (and 6 different physical stores);
- Data often contains values which are simply impossible (for example temperature of water at 125 ° at normal pressure) due to wrong sensor readings, necessary re-adjustments, and so on.

This data is very important as it is the basis for:

- Proof reports undertaken by the lab which verify a supplier's quality compliance;
- Production management reports which is extremely important for a manufacturer;
- Definition of quality criteria that suppliers have to obey when providing material treatment and coatings;
- Investment decisions, for example new paint finishing systems and their working parameters.

Currently data integration and consolidation are performed manually (using tools like Excel and Word) so all reports have to be processed manually as well. Only the templates for the reports are generated automatically.

Figure 3.1 shows the current situation of the corrosion lab process in UML by describing the logical flow of information between the components, which form the lab.

[39] Currently Daimler in in the progress of changing things to a device gateway architecture, so things improve and get better

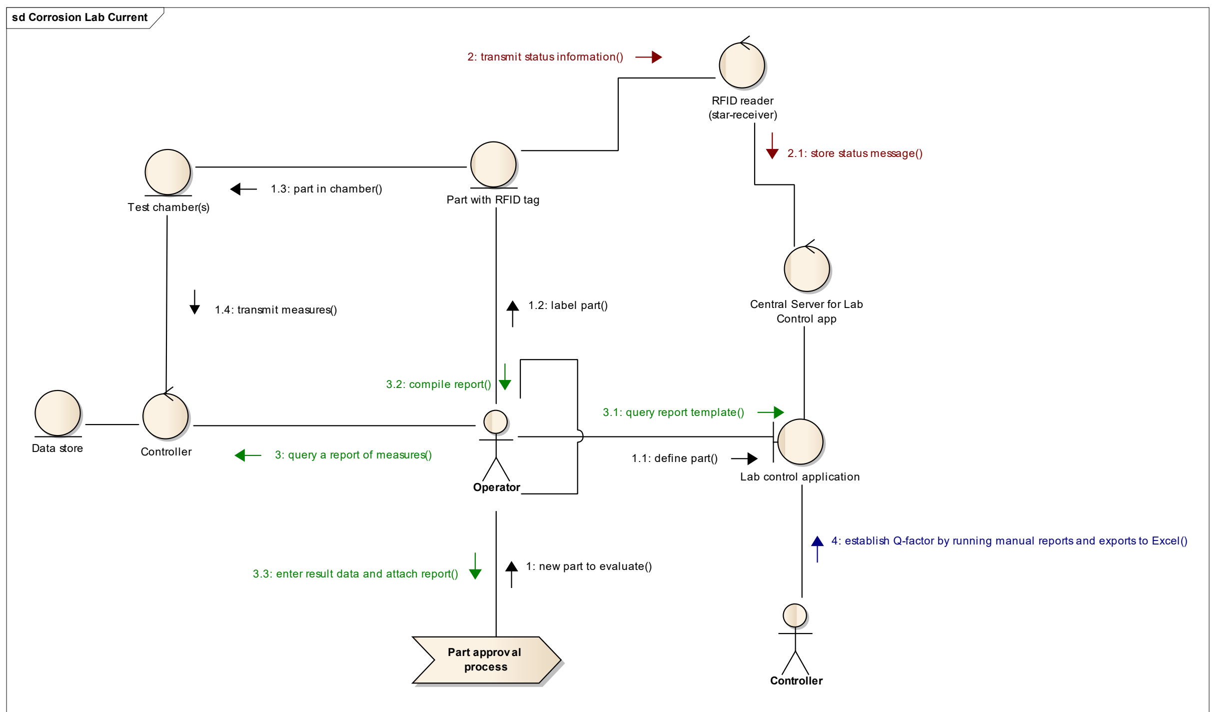

Figure 2.2: Current Corrosion Lab Process

The situation is even more exaggerated as some tests are run on separate equipment and produce additional data (in form of Excel files for the measures and images for the result) which currently is not automatically incorporated into the lab control system. As these usually are calibration tests performed with some 300+ scratch pads this is an enormous amount of additional data[40] which would need proper integration and analysis to be able to give better quality-factors (Q-factors) and early warnings for failure.

Given this integration example the business process and internal requirements of the author's customer have been evaluated in several workshops and requirement analysis documents bilaterally with the customer and the results are presented in the next two sub-sections.

2.5.1 Business process interests and requirements

From the business process side, the requirements towards the corrosion test center are:

- Integrate directly and automatically in the approval process of new parts which is very complex, time-consuming and expensive;
- As many manufacturers produce the same parts in the same way (and are purchased to maintain independence) it has to be ensured that the quality remains high and

[40] Each pad generates 2 images (each 4 MB) and 3 measures (longest thread, average and mean)

stable. Any fluctuations have to be immediately reported back so that very expensive callbacks and warranty operations can be avoided;
- If tests show significant problems of a material some recommendation towards production is required.

Interestingly these requirements have nothing to do with the device integration (chamber, part with RFID-tag) as such, but are simply using the results which such an integration could generate.

If the whole corrosion test center is considered a very complex device (which it is to some extent) then the integration into the overall production process would be much more streamlined and the efficiency (by reducing manual work) increased. This efficiency increase would be an especially important requirement from business process side as operational costs are high due to the amount of labor required.

2.5.2 Internal requirements

The corrosion test center has the following logical requirements which need fulfilling:

- Correlate all data so that the same scientific unit, data format and representation is used;
- Eliminate implausible data (many of the temperature readings are simply impossible) and therefore provide a much more reliable data basis;
- Store data in a centralized data store so it can be used in a uniform way;
- Provide unified access to data of the same type for comparisons;
- Be able to identify any part position over time for each part;
- Provide automated warning if parts are lost (not in any chamber, yet due for tests);
- Automatically adjust the workflows in the facility control software in a way that interrupts (cleaning, part disposition) are booked into the workflow;
- Be able to have for each part historic information as to when each measurement was retrieved from which chamber;
- Calculate a dynamic Q-factor (quality measurement of the lab) which combines accuracy of the tests, timeliness and compliance to requirements;
- Create part approvals automatically based on evaluation of measured criteria versus requirements from the norm specification.

Combining these logical requirements, the following requirements can be derived:

- Data quality has to be integrated and executed before data enters the business process;
- Data format has to be irrelevant;

- Communication interfaces and methods towards devices and consumers have to be irrelevant and flexible / exchangeable;
- Where the data is consumed (data target) is irrelevant;
- Data must be semantically enriched so added value is presented to the business process
    - Either by generating entirely new data like a Q-factor;
    - Or by combining data like what the temperature's measure for part X on a particular date was.

2.6 State of the art in device gateway architectures

Section 2.6.1 provides an overview of a number of key device gateway architectures from the state of the art. Section 2.6.2 compares the features (defined in Table 2.1) of these architectures against each other. The data protocols used in state of art are then described in section 2.6.3.

2.6.1 Overview of existing device gateway architectures

The following device gateway architectures (or methodologies to achieve a similar functionality) were selected as the state-of-the art architectures that should be described:

1) Xively (now Google Cloud);
2) MS Azure IoT (generally message bus-based systems);
3) sMAP;
4) Custom / Proprietary solutions.

They have been chosen for the following reasons:

- their coverage in academic publications[41];
- their market presence for commercial solutions[42];
- their use in real-world environments (based on the author's experience).

The list could be extended by many other solutions, most notably SenseWeb [7], TinyDB [79], SENSEI [11], IrisNet [81], SIRENA [89] or SOCRADES [90], yet all of those lack a larger

---

[41] For example, the sMAP paper (http://dl.acm.org/citation.cfm?id=1870003) at SenSys 2010 has been cited 153 times and there are in the region of 10 papers covering the topic; Azure IoT has references in 1,260 papers (according to Google Scholar as of 11.07.2016)

[42] Xively as part of Google Cloud IoT and Microsoft Azure IoT, are rather large

installed base or are no longer used / active (at least accessible to the author) and therefore were not included.

In addition to the above list, gateway architectures have been designed or suggested by institutional bodies like Internet of Things Architecture (IoT-A) [25], EPCGlobal [26], Open Geospatial Consortium [27], EPoSS [28] and others. Common to all these projects is their large scale and more fundamental and wider research scope (beyond the scope of just device integration part). Therefore, as the approach proposed in this thesis is more towards a direct and tangible improvement of the device integration part, these were not included.

Another group were not included in the analysis, as they are very similar to the chosen Xively and Azure IoT, namely platforms like Amazon's AWS IoT (https://aws.amazon.com/de/iot/how-it-works/) or Beckhoff's TwinCAT (http://www.beckhoff.com/twincat/). They provide more or less the same functionality and feature setand have a messaging-based approach with some additional processing.

Finally pure IoT gateways ([91] provides a good overview of available technologies) and pure industry automation device gateways were also not included in the analysis. This is because: IoT gateways tend to be WSN[43]-biased and focus mainly on issues associated with the runtime environment (power-efficiency, communication issues, edge- and fog-computing, etc); pure industry device gateways only tend to focus on the protocol support and interchange, like Soft PLC's[44] protocol converter. There is a tendency for overlap in areas of fog-computing and industry automation [98] yet this is not covered as well.

2.6.1.1 Xively (part of Google Cloud IoT)

Xively (www.xively.com) is a commercial[45] system whose components are shown in Figure 2.3. The services are running only on Xively systems, so no standalone or on-site installation is available which means that an active Internet connection is required all the time.

---

[43] WSN = Wireless sensor networks – where sensors are connected using wireless technology

[44] http://www.softplc.com/products/comm/gateways/

[45] Personal users can use it - within limitations - free of charge

Figure 2.3: Xively services

All data producers and consumers connect to the central system using a standardized publish-subscribe messaging transport protocol MQTT 3.1.1[46]. Connectivity via REST (over http(s) and Sockets) is available as well, yet more for compatibility with previous versions of the product than for emerging integration scenarios.

The internal payload of all telegrams (regardless of the communication link) is either JSON, XML (in EEML[47] dialect), CSV[48] or XLS[49] (for series data).

The design allows for an arbitrary number of consumers to be connected (logically linked) to an arbitrary number of producers (sensors / devices). So messages from one producer will be transported to the centralized system and there the broadcasting to a whole range of registered receivers (consumers) tales place using MQTT. This might be a reason why MQTT was given precedence over previous, more technical access protocols like REST and sockets, as it focusses on a more abstract level[50]. A message receiver can be a producer (thus a device) as well, meaning that sending data to the device is implemented by means of MQTT-messaging.

---

[46] http://mqtt.org/documentation

[47] EEML = Extended Environments Markup Language - http://www.eeml.org/. A XML specification to describe environmental data, developed by Xively while still being Pachube.

[48] CSV = Comma separated values

[49] MS Excel file format

[50] This is a fundamental feature of most messaging-platforms (like Microsoft Azure IoT as well) which might be a reason why IBM (using the know-how they gained from WebSphere MQ-Series) is participating in the MQTT consortium as well

Xively handles the messaging part (both ways –read and write) very effectively and by using MQTT (internally as well as externally) as an abstract protocol in a very device-independent way. The data hierarchy used by Xively is based on products which contain several (1..n) devices which have one feed. The feed is the collection of data streams (0..n).

In addition to just forwarding information from producer to consumer Xively can aggregate time-series data as well, which means they collect data from a producer over a period of time and make these available as a "time-series". The maximum duration of a time-series window which can be retrieved is 6 hours – in case a longer window is requested several requests have to be made. The same information collection and retrieval is offered for incident and status logging as "log-data" service.

Xively includes the ability to define triggers to notify registered listeners about special data constellations as well. Here the intended listener is registered for a feed with a condition which has to be met and in case the condition occurs the listener is called using a HTTP POST request. So, as an example, a listener could register for the feed on "device ABC" for the condition of data stream "Temperature Celsius" is >= 80 and in case this happens the HTTP POST to the registered URL will occur.

It is not defined, if the trigger is called synchronously or asynchronously to the occurrence of the event, and how much the delay between occurrence and call would be. Thus usage in a critical environment, where for example reaction to a threshold breach within x msec is important, is not possible.

So, in principle Xively is a large-scale message exchange with the core focus on very fast publishing of inbound data to multiple subscribers with an additional internal data store for historic data. Aggregated data as time-series and logs, in addition to triggers and optionally an integrated SalesForce[51] integration can be used as well.

This clearly classifies the product as a device gateway, yet compared to the proposed DBGA (see next chapters) some short-comings are clear:

- Xively is commercial and therefore everybody using it has to pay;
- Many people in the industry handling data would not be able to utilize the Xively service as their company policy prevents them from storing these data packets anywhere outside the company[52];

---

[51] SalesForce is market-leading Customer Relationship System (CRM) - https://www.salesforce.com/

[52] For example, at Daimler Benz AG any data which is considered relevant (which is below confidential) can only be stored in in-house data centers. If it is confidential further limitations apply.

- Xively can only be used as a centralized service requiring any submitting device or receiving application to be connected to the Internet which is very often simply impossible or not allowed (security, policy, etc.);
- The main target group of Xively data is more classical consumers like data visualization, CRM systems or central monitoring systems;
- Triggers are the only way a client can react to a change in data. No autonomous operations by the device gateway are defined or possible, thus making it very hard to implement a higher degree of control;
- Xively does not filter, alter, cleanse or operate on device data in any other way than time-series aggregation.

2.6.1.2 Azure IoT

Microsoft (MS for short) Azure[53] is a collection of cloud-based services offered by Microsoft (MS). Among these services the IoT-suite[54] acts as a complete IoT solution implementing Microsoft's IoT architecture [81] (where a device gateway architecture is part of it). It should be noted that the MS Azure IoT suite is a combination of several micro-services from the range of the MS Azure services with some pre-configuration applied. This level of functionality could be achieved by using and connecting the core components as well. In order to create a fully working system these components normally have to be adjusted and configured anyhow.

Figure 2.4 shows the schematic overview of the MS IoT solution architecture.

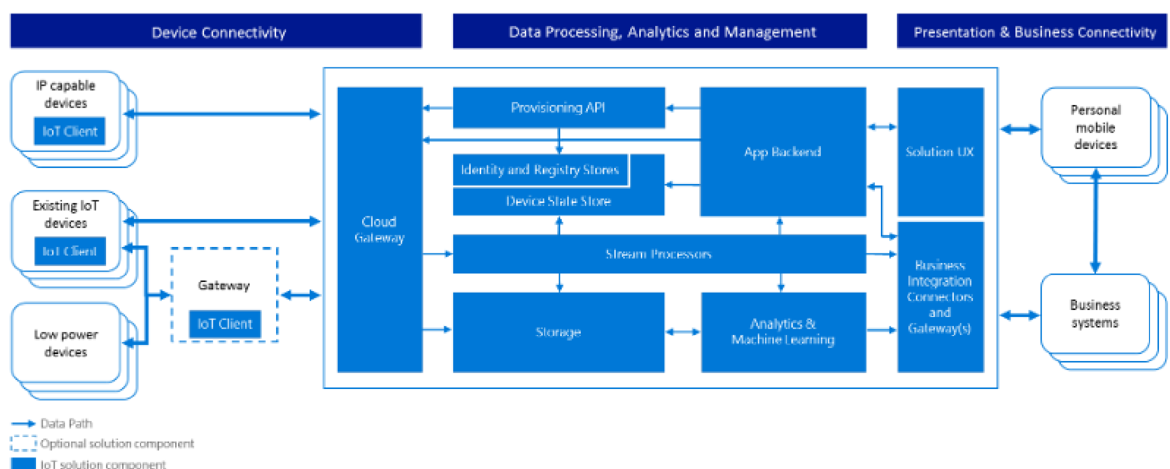

Figure 2.4: Azure IoT solution architecture [81]

---

[53] https://azure.microsoft.com/en-us/

[54] https://azure.microsoft.com/en-us/solutions/iot-suite/

For achieving the functionality depicted in the architecture several other services from the Azure range are combined. These combined services can be identified in Figure 2.5 which is an overview of the MyDriving[55] sample application (developed by MS to showcase their technology). From the figure the following main Azure services used can be identified:

- *IoT Hub* as a cloud gateway to provide the core connectivity from a device to the IoT service;
- *Blob* and *SQL database* to store data permanently;
- *Stream Analytics* and *Message Bus* to do the life data processing and eventing;
- *Machine learning* to automatically figure out new facts;
- *Monitoring* to keep track of ongoing operations;
- *Processing unit* to run several services.

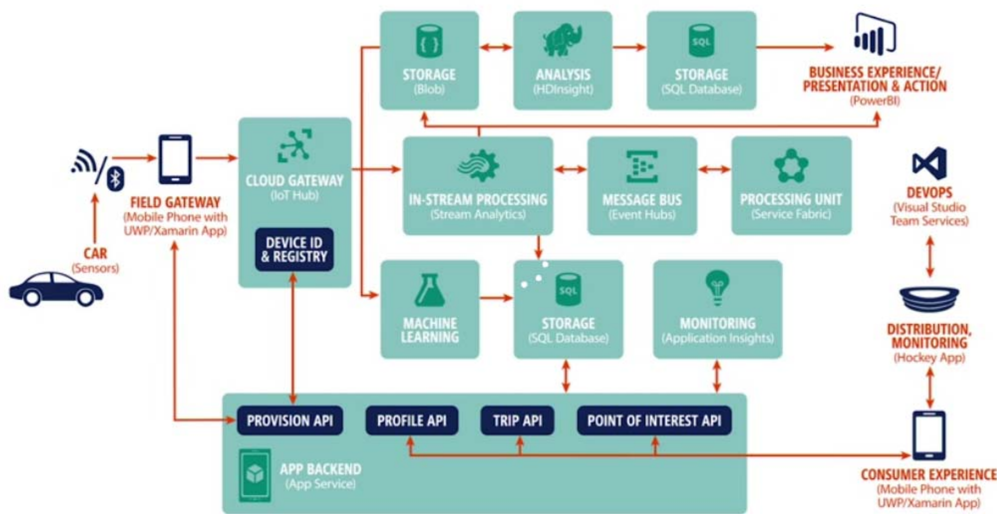

Figure 2.5: MyDriving sample application in Azure IoT showing sub-services used[56]

In this architectural approach many separate components (services) are combined to form an entire architecture and application landscape, which has an added benefit for MS insofar as several services have to be consumed (where each service has its own cost associated).

On the technical side the approach stems mainly from the fact that the predecessors of the architecture utilized the BizTalk message bus architecture as described in [9] and [10]. This

[55] https://azure.microsoft.com/de-de/documentation/samples/mydriving/

[56] https://azure.microsoft.com/de-de/documentation/samples/mydriving/

message bus is available as a separate hybrid messaging and application integration platform as well (Azure BizTalk services)[57].

At the very core of the architecture is a message bus – similar to the one used by Xively (see section 2.6.1.1 for details). So, both architectures operate internally on the principle that inbound messages are routed to dedicated receivers. The message content is of no concern to the routing engine as it just routes a data packet from A to B. The message format used internally is always XML to enable the various parties to access the payload in a homogenous way.

A device is considered a data stream source where conceptually a single data stream for all attached sub-components of the device (for example sensors) is utilized and each sub-component writes into the same stream. In this main stream the sub-components are identified by some property of the data record (for example "stream-id" or "sensor-id"). Any data packet sent in a message is just considered an anonymous data record in a data stream from a device which has a unique timestamp, so the data as such is of no concern to the gateway.

Any device in the MS architecture never accepts unsolicited network connections which means that all connections are initiated by the device as an outbound operation. Therefore, sending data to the device (for example writing a value or sending a command) will be implemented using sending messages to a queue or store for the device where the messages are persisted for guaranteed delivery. Once the device is active (again) the messages will be retrieved. To activate a device when it is in for example power-down mode (so that it retrieves the messages) an out-of-bound signal (like a SMS) might be sent to wake up a device and then the device can retrieve the pending messages. Messages are stored until the device retrieves the messages or the Time-to-Live (TTL) time for the message expires. This is important as some data (for example the command to open a flap on an exhaust) might only make sense within a specific period of time (for example next 2 minutes).

Connecting a device to the MS architecture can be done in several ways:

1) *Direct connectivity to the cloud gateway*
   Here the device is directly connected using a TLS/SSL stack and TCP/IP to Azure.

2) *Connectivity via a field gateway*
   Used mainly for devices using specific protocols, short-range communication

[57] https://azure.microsoft.com/en-us/services/biztalk-services/

technologies (like ZigBee or BlueTooth), devices not supporting a TLS/SSL stack, or devices which are not connected to the Internet. In addition, this option might be used when some sort of aggregation of streams and data shall be performed before the data enters the cloud (for example limit check).

Field gateways are different from mere traffic routers as they take an active role in managing access and information flow.

3) *Connectivity via a custom cloud gateway*
Custom cloud gateways are part of the cloud based architecture and thus operated in the cloud context as well (in comparison to the field gateway which – usually - remains outside the cloud). They generally manage all aspects of communication, including transport-level connection management, protection of the communication path, device authentication and authorization towards the cloud gateway. Quota enforcement, billing data collection, monitoring, etc. are all part of their normal operation.

4) *Combination of Field Gateway and Custom Cloud Gateway*
This option simply combines the previous options.

The actual communication protocols (in case direct connectivity is used as otherwise any protocol could be utilized) provided are: HTTPS, AMQP [58]and MQTT (as Xively). Using the provided libraries for connectivity available for .NET, Java, Node.js, Python or C (which can run directly on a device)[59] greatly reduces the implementation effort.

Compared to Xively, MS has put much more emphasis on the processing facilities of inbound data as well as further “higher” services like machine-learning and data analysis. Inbound data streams can be analyzed “on the fly” using stream processors or data pipeline transfer and the resulting data stream is then forwarded to the next handler. Using several handlers, a chain of handlers can be created where each handler will only do its part. Therefore, it is possible to completely analyze, process, change and even suppress incoming data before being handed over to the corresponding receiver(s).

---

[58] AMQP = Advanced Message Queuing protocol (http://www.amqp.org/) – a protocol like MQTT yet on a far more global scale

[5959] https://azure.microsoft.com/en-us/documentation/articles/iot-hub-sdks-summary/

As all services are run in a stateless manner in the cloud the architecture provides in principle limitless growth which clearly coincides with their architectural definition [81]: "The proposed architecture should support millions of connected devices. It should allow proof-of-concepts and pilot projects that start with a small number of devices to be scaled-out to hyper-scale dimensions."

Given that and the structure of the services which can be combined in flexible solutions, the MS architecture is for sure one usable for extremely large environments containing millions of devices. On the downside this comes with considerable administrative and integration effort. Especially as all the components have to be developed, deployed and operated which requires several areas of special and dedicated knowledge.

As well as for Xively the comments about being commercial and not available for on-site deployment compared to the proposed DBGA (see next chapters) remain. Yet as there are many options regarding stream processing and data handling, the architecture is quite similar to the DBGA (but with a different way of doing things). More detail on the differences is seen in the case study sections.

2.6.1.3 sMAP

sMAP (the Simple Measurement and Actuation Profile[60] - details at [5], [18] and [57]) was developed at Berkeley University to address sensing issues and sensor integration in building and environmental monitoring scenarios. The main intention was to represent physical devices (which usually have / had no TCP/IP or even HTTP stack available) to an outside world by means of HTTP resources.

Its core concept is that a physical device is represented in the HTTP space by a "sMAP source" (which is connected to the actual device using a driver). This source, which technically acts as a tiny web-server, is then available to be queried using RESTful APIs from anywhere using JSON-encoded data. In addition, consumers for a source can be configured and then the source sends HTTP POST to each consumer when data changes. Thus the source is a unique information object (resource) addressable using a URL in the whole range of other objects available.

sMAP is written nearly entirely in Python and available in source code as well. The current drivers are more towards building and environmental monitoring (power-consumption, temperature readings, humidity, etc.) whereas different types could be connected rather

[60] http://www.cs.berkeley.edu/~stevedh/smap2/index.html

easily. Receiving data as a client can be done with either "R"[61] or using sMAP's own query language (embedded in the JSON request).

Using sMAP's approach to create new sources (and connect devices to it) is rather fast and straightforward and allows any integrator to get some basic system up and running in a short time. However when you start having quite a number of sensors things start getting complicated with all the small web-servers around; especially discovery and security. Allowing an "intelligent" device to act as its own source by hosting a Python environment could be done, but due to the availability of Python in embedded environments this would not be very easy and especially involve quite some testing.

The sMAP adopted "source" approach is fine in a research context, where you could walk to a "source" which is not responding properly (typically a dedicated machine), but will not work so well in a more industry or automation environment. In addition, from a security point of view these tiny servers are sub-optimal and most industrial customers would simply not tolerate such an approach from the security side (controlling access, securing data, encryption, etc.).

Querying data using R or the sMAP-query language is a very interesting feature for scientists and people with technological background, yet the access from tools like Excel[62] (using for example ODATA) is far more important for many people and not implemented at the moment. Of course, you could build a connection (using R or some other technology) between Excel and the source, but still you have to do it and it is rather complex work.

The missing "centralized" coordination results in there being no option to combine several data-sources and create new knowledge out of this. You could achieve this, of course, by creating a new source which subscribes to other sources and provides new data based on the ancestral data. Yet still this has several disadvantages in that: it is not in centralized repository; has to be done anew for each constellation; no central management, and so on.

sMAP is more focused on data aggregation and then afterwards retrieval than active process control or data manipulation. In addition, sMAP provides no added services like actuators, triggers, data enrichment, etc. - all these features have to be a custom adaptation and implementation.

[61] https://www.r-project.org/

[62] Excel being the "tool of trade" for most users of these device data in a more aggregated form

2.6.1.4 Custom solutions (proprietary approaches)

Proprietary approaches, usually containing a mix from all above, are also in use (for example at the author's customer projects as well). Typical examples, which were examined during the state of the art review, are (among others): CTS Umweltsimulation[63], WUT Simpati[64] and Johnson Control Umweltsimulation. In these examples, the gateways bundle devices and act as backup control units (in case a centralized system fails and data has to be stored for some time before it can be re-transmitted). When a connection is there, any event is propagated up the chain to centralized systems where the final storage (usually in more or less proprietary data silos) takes place.

These solutions usually are quite performant and functional as long as no further process integration is needed and / or data from the processes required. Especially when it comes to dynamically generated values (like quality of service, throughput times, etc.) or additional services these systems usually will not support these operations / requests.

The downside to such approaches is that any evolution or change requires additional investments (everything has to be developed). Retaining knowledge of the integration is also another problem with such solutions as they are usually bound to an integrator whom to replace (sometimes mandatory as for example the company ceases to exist) is hard and sometimes nearly impossible.

In short, these custom solutions are very much like a bespoke suit. For some they can be absolutely perfect and provide everything needed for a very reasonable price. For others (especially when requirements change frequently) they can become a major obstacle hindering progress and far too expensive for what they deliver.

2.6.2 Gateway architecture feature comparison

To be able to compare the various features present in the analysed architectures discussed in section 2.5.1, Table 2.2 has been created to illustrate the core features and their coverage.

The features listed in Table 2.1 were chosen as they, from the vast experience of the author as an integration practitioner in the industry automation area, are the most relevant when it comes to actually using and integrating a gateway in the real world:

---

[63] https://www.cts-umweltsimulation.de/produkte/software.html

[64] https://www.weiss-technik.com/en/productarea/smpatiR-software/

Table 2.1: List of features

#	Kind of feature	Feature
F1	Functional	Available “in-house”
F2	Functional	Hostable in the cloud
F3	Functional	Support for very large number of devices
F4	Functional	Failover, Cluster-Support
F5	Functional	Data can be written to the sensors
F6	Functional	Asynchronous information about changes
F7	Functional	Autonomous system support / workflow operation
F8	Functional	Data is stored internally
F9	Functional	Inbound data can data be intercepted, analysed, checked?
F10	Functional	Support for data querying
F11	Functional	Communication pattern
F12	Functional	Synchronous / asynchronous communication model
F13	Functional	Level of coupling of gateway and consumers (client)
F14	Functional	Preservation of state (knowledge about a sensor)
F15	Functional	Semantic data value increase
F16	Functional	Ontology support
F17	Non-Functional	Customization required before use
F18	Non-Functional	Amount of infrastructure / additional services needed
F19	Non-Functional	Cost

For a full description of each feature chosen see B.1 in the appendix.

Table 2.2: Feature overview

#	Feature / Functionality	Azure IoT	Xively	sMAP
F1	Available “in-house”	No	No	Yes
F2	Hostable in the cloud	Yes	Yes	Maybe with a lot of work
F3	Support for very large number of devices	Yes	Yes	No
F4	Failover, Cluster-Support	Yes	Yes	Partially (database yes, gateway only cold-standby)
F5	Data can be written to the sensors	Yes (yet requires special integration work)	Yes	No
F6	Asynchronous information about changes	Yes	Yes (only via triggers)	No
F7	Autonomous system support / workflow operation	Yes, but not controlling the device directly	No	No
F8	Data is stored internally	Yes (with additional modules)	Yes	No (can be achieved with external services)
F9	Inbound data can data be intercepted, analysed, checked	Yes	No	Yes, but with effort

#	Feature / Functionality	Azure IoT	Xively	sMAP
F10	Support for data querying	Yes (database operations and analysis)	Somehow by means of web interfaces and small analytical parts	Yes, a specific query language is provided as well as support for R
F11	Communication pattern	Message based	Message based	API
F12	Synchronous / asynchronous communication model	Asynchronous (message based)	Asynchronous (message based)	Synchronous (API-based)
F13	Level of coupling of gateway and consumer (client)	Loose	Loose	Tight
F14	Preservation of state (knowledge about a sensor)	Only with additional services	Only partially as historic data	Only with additional services
F15	Semantic data value increase	Not directly – requires quite a lot of additional work	No	No
F16	Ontology support	With additional effort	No	No
F17	Customization required before use	Yes	No (but could)	No
F18	Amount of infrastructure / additional services needed	Much – many services and additional features have to be combined and integrated	Very little – almost all is hosted as one solution in the cloud with just a few options	Very little

#	Feature / Functionality	Azure IoT	Xively	sMAP
F19	Cost	Fixed cost is usage based, can be quite high[65]. Follow-up cost due to specialized skill requirements very high	Fixed costs are moderate – per customer specific deal. Follow up costs is moderate as well as there is not very much to configure	Free fixed costs (aside hardware). Follow-Up costs usually quite low

[65] Pricing is actually quite complicated as for example for an S1 edition (42 € / month) 400,000 messages / day (each 4 KB block is one message) is included. In case additional services like Machine Learning, App Service or Stream Analytics are used, these are charged on top.

2.6.2.1 Summary of feature comparison

As each feature can be more important than others in a specific integration environment, any kind of prioritization is not really possible. Yet from a current industry usage point of view, based on the author's experience, the most important features are:

a) The preservation of state (so the knowledge how the current sensor state is);

b) Semantically enriched data which can be utilized by a business process to be re-used higher up in the value chain;

c) Flexibility to support synchronous as well as asynchronous communication patterns, loose or tight coupling between consumer and gateway and the ability to use an API instead of a message-based approach;

As in-house[66] message bus operations become available[67] and are being utilized, protocols like MQTT (see Table 2.3) and asynchronous principles become more important in general. This is very much in line with the current trend to utilize the Internet as a communication media for industry data as well ([84], [106]), especially when the value-chain and data model stretches over several different companies (for instance suppliers and consumers). This can be seen in solutions like Xively or Azure IoT as they provide this as their only operational model.

2.6.3 Data protocol support

Data protocol support is a key aspect of any gateway architecture (indicated in [106] as well). In general data protocol support is very heterogeneous among the different architectures with REST being supported by all, yet on different levels. For sMAP it is the only option to communicate with the nodes, for Xively and Azure IoT it is mainly

---

[66] Which is an important difference for most users as usually no sensitive production data will be allowed outside the premises

[67] For instance, in Daimler AG (and several other large manufacturing sites in Germany) there is currently a process to establish a factory wide message bus for any kind of production related data

used to manage things (devices, streams, etc.) and for the DBGA all operations are available via REST.

SOAP is currently supported by none of the architecturs, yet in case of need a custom cloud gateway could offer the same functionality for Azure or in a similar manner for Xively.

Azure IoT exclusively provides AMQP as native internal protocol, whereas both Xively and Azure offer MQTT for message transport.

To standardize efforts for integrating industry automation devices the protocol OPC-DA [86], later followed by OPC-UA [85], has been developed by the OPC. It is the internal data transmission format over any external link format like REST or SOAP, and thus defines a common format for the exchange of data and information retrieval. It is used mostly in SCADA- or ERP/MES-system-integration but can be utilized by any consumer. Usage examples can be found in [99].[98]

The protocol support matrix is shown in Table 2.3:

Table 2.3: Data protocol support

Protocol	Azure IoT	Xively	sMAP
REST	Yes, but Throttling may be activated[68]	Yes	Yes
SOAP	Only with custom implementation	No	No
AMQP	Yes	No	No
MQTT	Yes	Yes	No
Binary protocol	Only with custom implementation	No	No
ODATA	Only with custom implementation	Only with custom implementation	No
CNDEP	Only with custom implementation	Only with custom implementation	No
OPC-DA / OPC-UA	Only with custom implementation	Only with custom implementation	No
Data content agnostic	Yes	Yes	No

Azure IoT custom protocol implementations can be done by implementing an IoT hub[69] where the protocol conversion to and from AMQP is performed.

Custom solutions cannot be classified as specific from use case to use case. Yet usually their protocol support is limited to that which is absolutely necessary, which in most cases known by the author is either a binary protocol using native TCP/IP frames or REST services. In more automation industry related solutions SOAP is predominant, as REST is as yet not considered deterministic or predictable enough.

---

[68] (https://azure.microsoft.com/en-us/documentation/articles/iot-hub-devguide/#throttling) based on communication pattern used.
For example, for receiving messages from the cloud the supported frequency using HTTP/1 is 1 request per 20 min (per device) - https://azure.microsoft.com/en-us/documentation/articles/iot-hub-csharp-csharp-c2d/

[69] https://azure.microsoft.com/en-us/documentation/articles/iot-hub-devguide/

Current gateways (aside from custom solutions) all support using JSON data in a HTTP request.

2.7 Discussion

Information about devices and sensors (section 2.1), industry automation (section 2.2) and Industry 4.0 (section 2.3) and the place of device gateways in automation and Industry 4.0 (section 2.4) was presented, followed by a real-world integration example from the author's customer in section 2.5, leading to a state of the art overview of existing device gateway architectures in section 2.6.

Based on this information, the presented architectures (see section 2.6.2), research papers like [101], [95] and [104], and various discussions with business stakeholders in the automation industry lead to the identification and confirmation[70] of several key criteria for device gateway architectures:

- Flexibility offered by the architecture for an integrator to integrate devices into business processes, including ease and cost of change in the future
  The complexity of the solution and the skill required to execute the solution, including the learning curve to obtain the skill level required;
- How much time (and therefore cost as well) it takes to:
    - build a solution for a use case based on the architecture;
    - Operate and administer the solution in one year;
- Simplicity of the design and the ease of change (including the cost of change) in the future including the preservation of investment;
- Easiness to obtain the current device / sensor state and use it in the business process;
- How much infrastructure and overhead in terms of "environment" is needed (footprint of the system) and what performance with which maximum number of devices / sensors / actuators supported is provided on this infrastructure;
- How reliable and fault tolerant is the system and what is the overall stability;
- Is it free software or a commercial system, which costs are involved;
- What security mechanisms are in place.

These criteria define if a device gateway architecture is usable and adding value in the automation environment, and therefore will form the basis of further requirements

[70] These criteria were confirmed as reasonable Mr. Robert Mayrhofer, Head of Corrosion Testing Team, Daimler AG

(see section 3.3). Characteristics are derived from these requirements (see section 3.5), as well as measurements (see section 3.7) of how much achievement an architecture provides (for the results see the architecture comparison in section 7.3).

Based on the feature comparison (see Table 2.2 in section 2.6.2) the current systems (especially Azure IoT and Xively) are powerful with regards to business processes, yet lack the necessary features of preservation of state and the semantic enrichment of data (which is of paramount importance in industry automation environments) without extensive additional efforts.

Autonomous system support, is another very important feature when a device gateway is to interact directly with a device in a non-supervised mode, and lacking from Xively and sMAP. Here Azure IoT provides some support due to its very good and powerful workflow capabilities, yet lacks in the interaction with the device, which limits its usability.

Flexibility is to some extent present in all architectures, yet only in a quite narrow pre-defined set of operational limits, which does not provide much choice for integrators. Yet as flexibility very often is a driving factor in achieving an improvement in an integration of a device gateway into a business process (by having a free choice of communication patterns, deployment options, API style, message-based operations, etc.), this becomes more important.

Therefore, an architecture optimized to improve the integration needs to address all these issues (among other requirements) in its design.

3 Requirements and characteristics of a device gateway architecture

In this chapter descriptions of typical usage scenarios for device-gateways (section 3.1), which shows potential uses of the architecture in real life (and evaluation environments), as well as a description of standard use cases (section 3.2) are shown.

Based on these, together with features from the state-of-the-art section 2.5, the derived requirements that should be supported by gateway architectures are presented (section 3.4) and ranked (section 3.5). The characteristics of the proposed gateway architecture DBGA are then defined (section 3.5). Finally, the coverage of the derived requirements by the selected state of the art gateway architectures (including DBGA) are then compared in section 3.6 and the measures to define overall business process integration achievement are defined in section 3.7.

3.1 Usage scenarios of the architecture

To have as wide as possible spectrum covering a majority of typical situations, two scenarios for later case studies and evaluations of the architecture in chapter 7 were chosen which resemble typical environments for a device gateway architecture:

- A stripped-down version of a climate chamber control system (section 3.1.1) for use inside a corrosion lab including RFID integration very similar to the author's customer environment[71]. The current "real" environment is described earlier in section 2.5;
- An exhibition visitor congestion display system (section 3.1.2) which tracks users and shows a heat map[72] for congestion in the exhibition space.

In general, it should be noted that the scenarios are not complete. There are several special conditions and real-world requirements which cannot be met. Yet still the scenarios demonstrate a basic usage pattern and implement all needed technology and control which is required from a device gateway integration perspective.

Typical examples for conditions and requirements which are not met, including the work-arounds used in the case studies, would be:

- No physically existing RFID systems and test chambers were present in evaluation, so simulators to generate the events and data were used;

---

[71] For a real life version many more devices would have to be supported and much more statistical data retrieved or calculated - yet for the scenario all required parts are covered.

[72] http://www.businessdictionary.com/definition/heatmap.html

- In the real world, the chamber min / max values would be dependent on the current test and test cycle running as well as an aging factor, as over time (due to crystallization) readings especially for temperature and humidity are less accurate (and have to be compensated);
- A real-world system would operate with 2,000 – 10,000 parts at any given moment of time. As this is simply a scaling up of the data sets the operations were performed with a limited amount of 100 sample parts;
- Instead of real mobile devices a simulation process was used and 100 concurrent users, operating in a defined rectangular area, simulated.

3.1.1 Corrosion lab scenario simulation

Using a subset of the environment described in section 2.5 a scenario for a case study in section 7.1 is defined.

For the scenario the following requirements are given:

The chamber acts as a device with 3 connected sensors so it can be read from. A reading should occur every second, minimally every 10 seconds to guarantee proper values. Each value must be associated with a precise timestamp. Each sensor value has to be checked for current min / max constraints.

The exact location must be retrievable for each part[73] (based on the id).

An alarm has to be raised by writing to an actuator in the following cases:

- When parts are moved out of a chamber into the pre-chamber-zone and not restored into a chamber within 15 seconds[74]
- When one of the chamber sensor values is invalid or not available for more than 10 second (would be 15min in real life)

For each part a Q-factor[75] has to be calculated dynamically which defines how well the part is tested compared to the test specification. As this would be a rather complex operation in a real environment for the scenario it is abbreviated to generate a random Q-factor for each individual part.

---

[73] So, the last known position from the RFID scanner should be taken

[74] In real life this would be 1h. Happens when a chamber cleaning (has to take place sometimes during tests) either takes too long or parts are simply forgotten to be restored

[75] quality factor from 1 to 10 as highest

In the UML Use Case diagram shown in Figure 3.1 the standard use cases provided by the architecture as well as specialized versions needed for this actual scenario are defined.

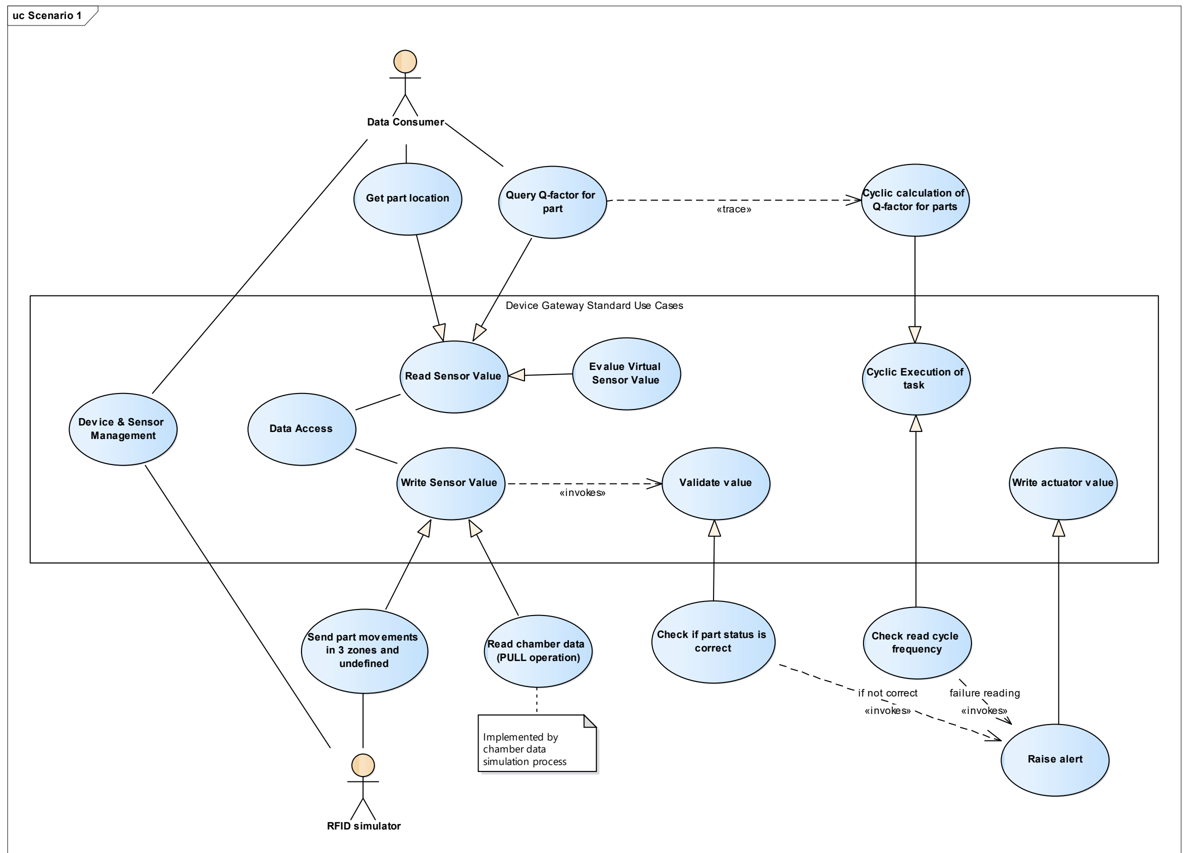

Figure 3.1: UML Use Case diagram for Corrosion Lab

In Table 3.1 the specialized use cases are defined in short (full details can be found in section A.1 (appendix)):

Table 3.1: Specialized use cases for Corrosion Lab

Use Case	Meaning
Get part location	Get the current zone[76] where the part (identified by its id) is currently located. This can be either the last transmitted position or the last stored value (needed after restart of gateway) or undefined in case no position is available

[76] Zone is a predefined symbolic name for a unique coordinate area. The RFID-infrastructure organizes all actual coordinates, derived by triangulation of the signal, into zones which are then communicated (instead of the real X/Y/Z position)

Query Q-factor for part	Get the current Q-factor for the part (identified by its id). If no calculation is possible (for example no data) 0 (lowest) is returned
Cyclic calculation of Q-factor	Perform a cyclic evaluation of all Q-factors for all currently active parts and write the results back to the gateway so that the next query will deliver the correct Q-factor
Send part movements in 3 zones and undefined	Simulate sending movement information for the parts into one of the 3 defines zones A, B or C or as undefined.
Read chamber data	Scan the defined chambers (simulation process) and retrieve the readings for the sensors
Check if part status is ok	Check if the part is not in an invalid chamber for the current test, the part is not scheduled to be tested, the part has been tested already or the part has been “forgotten” in the preparation area
Check read cycle frequency	Check that chamber data has been read and is valid within the last 15 minutes
Raise alert	In case an invalid condition is existent a global alarm has to be raised by writing to an actuator (which calls a WS* service)

3.1.2 Exhibition visitor congestion display system scenario simulation

As an alternative scenario for a case study an exhibitor congestion display system is defined in section 7.2.

For the scenario the following requirements are given:

When the user's position is sent to the device gateway it must be validated before it can be used (bounds check). Using the data, a clustering can be calculated by a centralized autonomous operation (workflow) every 1 second and the result as heat map data is stored. This data, consisting of an array of X/Y-coordinates based on a 10x10 cell size (thus dividing the coordinate space in 100 areas) and the number of people in the area will then be made available via a virtual value (data read request) and returned in JSON format so it can be consumed by the client to display the map. Using the heat map the distribution of people in the coordinate space can be shown using bright colors for many people and darker colors for less (or zero) people. Thus, the brightest areas would be those where the most people are.

In the UML Use Case diagram shown in Figure 3.2 the standard use cases provided by the architecture as well as specialized versions needed for the actual scenario are defined.

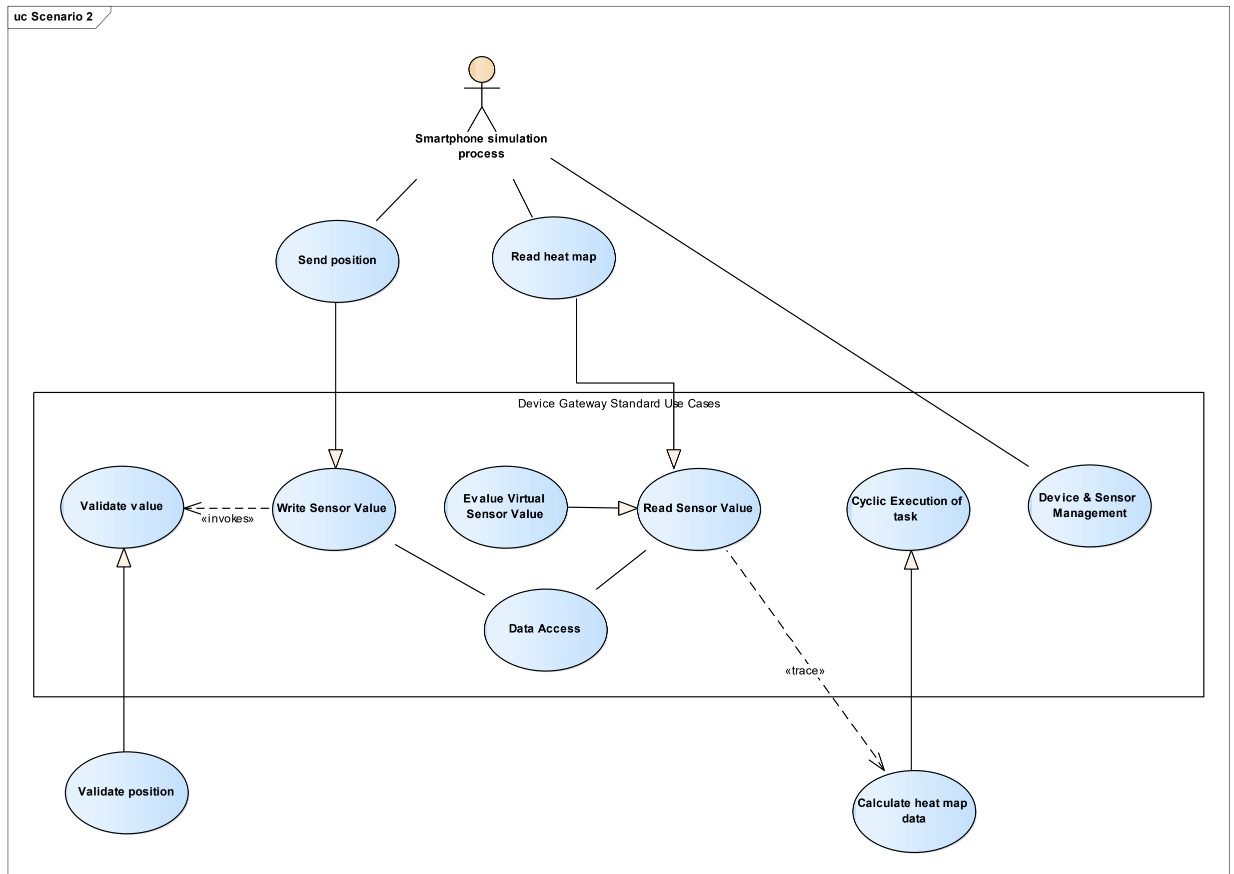

Figure 3.2: UML Use Case diagram for exhibition visitor congestion display system

In Table 3.2 the specialized use cases are defined in short (full details can be found in section A.2 (appendix)):

Table 3.2: Specialized use cases for exhibition visitor congestion display system

Use Case	Meaning
Send position	Send the simulated position in a 100 x 100 rectangle. Sometimes random errors (too large or too small numbers) are sent
Read heat map	Read the current heat map data as JSON array
Validate position	Make sure the coordinates match the rule, otherwise ignore
Calculate heat map data	Take the current valid positions of all participants and calculate the heat map data

3.2 Standard use cases to be supported by a gateway architecture

Analysis of the scenarios outlined in section 3.2 plus knowledge of other real-world scenarios have been used to derive a set of standard use cases that a gateway architecture used for integration into other systems should support:

- *Manage devices and sensors (details in section 4.5.1)*
  Consumers can register, query, delete and update devices and sensors attached to devices. This includes virtual devices as well.
- *Write sensor value (details in section 4.5.3 and 4.5.4)*
  Data producers use this use case to be able to deliver new sensor values into the system. This can be a push or pull operation – so either a write to the gateway by devices or a read from devices by the gateway.
- *Read sensor value (details in section 4.5.5)*
  Data consumers use this use case to be able to get existing values from the system.
- *Evaluate virtual sensor value (as a specialized form of "read sensor value") (details in section 4.5.7 and especially 4.5.7.4)*
  When a sensor is defined as virtual value the evaluation the read operation is done by the evaluation.
- *Write actuator value (details in section 4.5.2)*
  Allows the gateway users to be able to write values to an actuator.
- *Validate value (details in section 4.5.7)*
  The gateway uses it to validate (and potentially reject or adapt) a value before it is accepted into the system.
- *Cyclic execution of tasks (details in section 4.5.7.5)*
  The gateway uses it to run periodic tasks automatically.
- *Data access (details in section 4.5.6)*
  Used by write and read sensor value to have access to a core data storage functionality for reading / writing sensor data, etc.

3.3 Requirements to be supported by a device gateway architecture

The functional and non-functional requirements (section 3.3.1 and 3.3.2 respectively) of a device gateway architecture were derived based on the use cases and the state of the art review (section 2.6) as well as key criteria presented in section 2.7 and academic papers [84], [102], [104] and [106] among others.

3.3.1 Functional requirements for device gateway architectures

Table 3.3 defines functional requirements derived from the discussion for device gateway architectures.

Table 3.3: Functional requirements for device gateway architectures

#	Requirement	Description
R1	Centralized data store	Data from sensors must be stored in a suitable centralized storage media (usually a SQL database).
R2	Data retrieval using query facility	The architecture must provide a query / data retrieval mechanism in the protocol
R3	Direct data retrieval from data store	Data from the data store should be retrievable directly by a consumer without interference of the gateway
R4	Communication interface agnostic	Communication interfaces must be exchangeable and may not hinder any interaction. No specific features only in one (or several) implementations and not in others should be allowed (for example operation X works on protocol A, but not in protocol B)
R5	Data format agnostic	Store data in an internal format which is not specific to any CPU and implies no meaning to the stored data
R6	Integrated data quality control	Provide a flexible, extendable internal data processing chain which adjusts and / or filters data before it is stored / forwarded
R7	Semantic data value enrichment	New data, created or derived based on scripts or definitions and, based on existing one, must be generatable
R8	Autonomous operation / workflow execution	The ability to execute workflows should be integrated in the architecture so that they can directly and independently react to external events
R9	Actuator support	Data must be capable of being written as well as read (if supported by a device)
R10	Write-target agnostic	Provide writing of values to either physical or virtual actuators (which could be for example a URL where the value is POSTed to). Using this feature allows to easily “chain” messages from a device gateway to the next system by simply writing to a, for example, URL and thus passing the message further on. If not present any chained system must listen actively for

		messages and retrieve and process them which would not be possible with for example a WS* as they by definition are passive and have to be activated / called.
R11	Provide feedback on data change	Inform any registered receiver about changes in data monitored without the need to constantly query the value (busy polling)
R12	Data Source Interaction agnostic	The architecture should support PUSH or PULL operations for its attached devices. For PUSH the device sends actively, for PULL the architecture queries the device
R13	Provide discovery support for devices / sensors	It must be possible that devices with a certain characteristic can be found
R14	Authenticate users and limit access	Access to the gateway must be limited to specific authenticated users and individual access rights have to be granted
R15	Preservation of sensor state	The current state of a sensor must be accessible at all times from outside so that the application does not have to maintain this state information itself.
R16	Ontology support	The architecture should have at least some basic ontology support to be able to retrieve sensor and virtual sensor metadata and associated information

3.3.2 Non-Functional requirements for device gateway architectures

Table 3.4 defines non-functional requirements derived from state-of-the-art review and discussed scenarios for device gateway architectures:

Table 3.4: Non-Functional requirements for device gateway architectures

#	Requirement	Description
R17	Architecture should be implementable using "free" software	Any software needed to build the architecture should not involve license fees to be paid
R18	Extendable and easy to adapt to changing requirements	The architecture should be based on pluggable components so an extension / exchange of a feature / functionality / communication protocol can be easily achieved, thus an integrator is able to react with agility to changing requirements

R19	Based on modern software engineering practice	Assert that principles of modern software design are incorporated and used in the gateway architecture
R20	Operations support	Provide functionality so that the operations of the architecture can be tested, tracked, traced and analyzed during runtime
R21	Integration of a new device should be straightforward	The architecture must allow that a new device can be integrated in a very short amount of time
R22	Provide scalable and fault tolerant infrastructure	The architecture should allow to be run on highly available system so that in case of failure a cluster failover can happen

3.4 Requirements rating

As input into the definition of the desired characteristics of the proposed architecture DBGA (see section 3.5), the requirements defined in section 3.3 can be rated for their importance with 3 as the most important, and 1 as the least.

A 'rating of requirements 'exercise for such gateway architectures very much depends on the actual environment a gateway architecture is to be targeted for.

For example, in case someone is planning to build a publicly available IoT system with hundreds or thousands of users, then security and access control would be of paramount importance. In case of a localized solution in the industry automation environment with private networks, usually sealed off by firewalls from the rest of the world and only accessed by some users, such a security requirement would have a lesser important role (yet the database access still is mandatory to be secured).

The author's area of experience (industry automation and business process integration) was used as foundation for the ratings assigned in Table 3.6 below, in combination with customer feedback and discussions about usage of device gateway architecture targeted for industry automation and business process integration environment. A rating of 3 indicates a most important, 1 a less important characteristic;

Table 3.5: Requirements/Characteristics rating for Industry Automation/Business Process integration

#	Requirement / Characteristic[77]	Rating (1-3)	Reason for rating
R1	Centralized data store	3	Centralized data is paramount as very often data from very diverse sensors must be compared and used
R2	Data retrieval using query facility	2	Historic data retrieval usually is not very often used during live control of a sensing environment but more a monitoring feature to investigate a for example anomaly
R3	Direct data retrieval from data store	3	Especially when using data analysis tools (foremost Excel using ODATA) or in case business data warehouse has to be supplied from the sensor data (perhaps aggregated values) a direct database access must be present to facilitate the load and speed requests.
R4	Communication interface agnostic	3	Different levels of API-support on different protocols is something very problematic as it requires constant checking which option is supported in which protocol.
R5	Data format agnostic	3	For many business integration scenarios data must be stored in for example JSON arrays, strings, state information so agnosticism is mandatory
R6	Integrated data quality control	3	Having a gateway-side quality control built-in (or easily to integrate) is very important as sensor data very often is faulty and erroneous
R7	Semantic data value enrichment	3	Most sensing environments will have the need to enrich data based on internal or external values so that the consumer can utilize the centrally enriched data without the need to re-invent the wheel
R8	Autonomous operation / workflow execution	2	Autonomous operation is a useful feature, yet if not present it can usually be substituted with an external application which reads the underlying data and reacts accordingly. This is not as good as an integrated support, just a remedy
R9	Actuator support	3	Supporting actuators depends very much on the usage scenario the gateway is supposed to support. As writing values to physical actuators is usually not done by PCs or monitoring applications directly, but by specialized industry controls (PLCs), the

[77] To simplify the usage, requirement and characteristic naming were harmonized to share the same naming.

#	Requirement / Characteristic[77]	Rating (1-3)	Reason for rating
			actuator then is a way to inform the PLC that something should be written. Therefore, the PLC becomes the target of the write operation and from then on controls the actual, physical write.
R10	Write-target agnostic	3	In combination with actuator support it is very important that the target of a write operation can be anything which is addressable. So either a physical device, or a logical address (for example URL), which could be used for data-chaining
R11	Provide feedback on data change	2	Calling back consumers about changed data is sometimes a useful feature, yet very often there are technical issues to overcome (like for example sending data back to a browser or mobile application is non-trivial). Therefore, this feature is not considered essential.
R12	Data Source Interaction agnostic	1	Being able to read as well as (wherever possible) write to an attached device is of paramount importance in any environment which tries to control any kind of automation process. If only reading is provided than the DBGA can only be used to enrich the existing data, but cannot actively control the process. Therefore, the architecture should support PUSH or PULL operations for its attached devices. For PUSH the device sends actively, for PULL the architecture queries the device.
R13	Provide discovery support for devices / sensors	1	Usually, discovering new devices / sensors is only necessary when they are not known beforehand by consumers. Even when the device is not known (for example mobile user transmitting sensing data) it is usually not necessary for a consumer to figure out additional data about the device aside its unique id which might be used to look up user data.
R14	Authenticate users and limit access	2	When working in public available gateway environments authentication and access limitation is paramount as data integrity may never be compromised. When working in clearly sealed off environments like in the industry (where sub-nets are totally isolated from any external access) then this is no real issue. It all depends on the nature and sensitivity of the data as well as the data policy in the environment.
R15	Preservation of sensor state	3	If the gateway is just used as a message routing engine from devices to consumers this feature is not relevant, yet if the gateway acts as a sensor state memory then it is very

#	Requirement / Characteristic[77]	Rating (1-3)	Reason for rating
			important to always have a valid sensor state (even if defaulted). As the usual environments where the DBGA is to be used needs such a state this feature became dominant
R16	Ontology Support	1	The need for ontology support usually only arises in environments, where the number of sensors and virtual sensors exceeds some usable limit or where logical grouping of data becomes important (for instance to retrieve all sensors associated with temperature). Therefore, ontology could be important, but usually it is no core requirement.
R17	Architecture should be implementable using "free" software	2	In a low-budget or academic research environment this is a huge issue, for commercial use, less. Yet the big side-issue with it usually is, that non-free gateways nowadays operate in the cloud which contradicts most business customers' requirement for having the sub-nets, where the data is sensed, disconnected from the Internet.
R18	Extendable and easy to adapt to changing requirements	2	Especially in sensing change is constant as the processes and infrastructure is in constant move. Therefore, adaption is important
R19	Based on modern software engineering practice	1	As long as the solution is operational and fully working, extendable and usable this is an important but not mandatory feature
R20	Operations support	1	Operating any system requires considerable effort (either if hosted by an explicit operations department or if a solution is hosted only internally in a lab). Therefore, support for this is a very useful feature to allow for smooth and easy operation.
R21	Integration of a new device should be straightforward	2	As the adaption requirement is about change, new device integration is important
R22	Provide scalable and fault tolerant infrastructure	2	Depends on the usage scenario. Very often a cold standby is sufficient, sometimes 24x7 operations are relevant. The gateway should provide some defined way of how scalability and fault tolerance could be achieved, but does not necessarily have to provide it directly.

3.5 Characteristics of the proposed device gateway architecture - DBGA

Based on the ratings of the requirements as described in section 3.5, it was decided that the proposed gateway architecture (DBGA) be designed to have the characteristics shown in Table 3.6:

Table 3.6: Table of characteristics

#	Characteristics
C1	Centralized data store: Store any sensor data in a centralized data store
C2	Direct data retrieval from data store: Provide direct access for 3rd party to the data store so they can read data directly
C3	Integrated data quality control: Provide a flexible, extendable internal data processing chain which adjusts and / or filters data on its way from data source to data destination
C4	Semantic data value enrichment: Provide script- and definition based "virtual sensors" which combine and / or process internal and external data sources to form "new" data
C5	Communication interface agnostic: Provide a wide variety of (extensible and pluggable) communication interfaces so that consumers can access and producers deliver data in the most appropriate methods, protocols and formats for them
C6	Autonomous operation / workflow execution: Allow the architecture to control and monitor sensor state changes and integrate the execution of workflows so the gateway can directly and independently react to external events
C7	Actuator support: Allow data to be written to end-points (which could be a data chaining as well)
C8	Write-target agnostic: Provide writing of values to either physical or virtual actuators (which could be for example a URL where the value is POSTed to)
C9	Data format agnostic: Store data in an internal format which is not specific to any CPU but in a human readable format (string)
C10	Preservation of sensor state
C11	Extendable and easy to change: Provide an architecture which is based on pluggable components so an extension / exchange of a feature / functionality can be easily achieved, thus an integrator is able to react quickly and effectively to changing requirements

C12	Provide feedback on data change: Inform any registered receiver about changes in data monitored without the need to constantly query the value (busy polling)
C13	Basic Ontology-Support: Provide a basic ontology level on top of the sensor information level so that information retrieval based on logical attributes and context can be performed effectively

The mapping between the requirements and the corresponding characteristic is detailed in Table 3.7:

Table 3.7: Mapping between characteristic and requirements

#	Characteristic	Requirement
C1	Centralized data store	R1
C2	Direct data retrieval from data store	R2, R3
C3	Integrated data quality control	R6
C4	Semantic data value enrichment	R7
C5	Communication interface agnostic	R4
C6	Autonomous operation / workflow execution	R8
C7	Actuator support	R9
C8	Write-target agnostic	R10
C9	Data format agnostic	R5
C10	Preservation of sensor state	R15
C11	Extendable and easy to change	R19
C12	Provide feedback on data change	R11
C13	Basic Ontology-Support	R17

These high-level characteristics are further detailed and defined in the following subsections.

3.5.1 Centralized data store

Having a centralized data store allows to combine data from sensors from different devices (which might come from different data generation tasks) regardless of their origin. This is very useful when an overall reporting / data analysis or data integration with further systems (like for example business data warehouses) is anticipated or necessary.

Usually a centralized data store could be a SQL database yet any form of central store is valid, as long as all participants can access it. As sensor data very often just contains huge amounts of small payloads (the actual values) a NoSQL database[78] like for example MongoDB[79] might be superior to a SQL-DB as much less control and structural information is needed.

In addition, as sensor data is usually just written upon arrival (and very seldom deleted or updated), little demand for transactions exists.

3.5.2 Direct data retrieval from data store

Especially when the raw (or only marginally, by means of query language) modified sensor data is needed in further systems like for example data warehouses, then a direct data retrieval is necessary. Otherwise all requests for sensor data would have to go via the gateway which simply, given long or complex enough requests could easily overwhelm the query facilities.

The access is always only a data retrieval (read) operation – writes and updates can only happen via the gateway to not compromise data integrity.

Usually data retrieval would be done in for example SQL or any other query language (like R), which might be usable to extract the data from the central data store.

3.5.3 Integrated data quality control

The internal data processing chain must be flexible and extendable when filtering data on its way from data source to the internal data representation. Flexible here means that the different steps in the chain can, but do not have to be implemented. These steps would usually be checks that the data:

- is valid as such (data format, precision) – syntax analysis
- is within the defined bounds (min / max) – band pass filter
- fits / matches the previous values (sudden surges, drops)
- is continuous (if a sequencing is possible)

Any implementer of the gateway should be free to decide if the default (which can be no implementation or an implementation based on a configurable default) or a custom

---

[78] http://nosql-database.org/

[79] https://www.mongodb.com/

implementation for processing step should to be taken. This way extendibility would be guaranteed.

Extendibility should be done in a way which allows a new behavior to be plugged into the system by simple configuration. The implementation as such should be doable in the most appropriate form for the given task – so in whatever language / environment.

Another requirement towards the value chain processing would be that in different stages of the chain external plug-ins can be called. These stages, when an optional exit to an external module should be possible are:

- new value arrives
- value is checked
- value is to be considered the actual value
- value is to be persisted
- value has been persisted

When an external module is called additional behavior (like for example protocolling of value changes) could be realized.

3.5.4 Semantic data value enrichment

As one of the core requirements is the generation of semantically enriched data the creation of such virtual data endpoints (device / sensor) must be easily possible.

These virtual endpoints must behave like physical endpoints, thus supporting to be persisted, having default values, a value history and taking part in the value chain processing. They must allow to be used as the basis for further endpoints as well – so virtual endpoint chaining has to be possible. This way a virtual endpoint A and virtual endpoint B, together with a physical endpoint C could form a new virtual endpoint Z, which might be the sum of the three other endpoints.

The way a virtual endpoint is defined should be easy and straightforward. As it implies a calculation of some kind (otherwise it would not make any real sense to have an endpoint) this calculation must be either performed in an internal scripting language of some kind or an external module. In case of an external module this should follow the same criteria as usual extension points (pluggable modules) for the for example value chain processing.

An additional requirement, in case an endpoint is defined as based on other endpoints (regardless if virtual or physical), would be to optionally update the endpoint when the

underlying endpoint(s) change. This way a data change in an underlying endpoint could then trigger a complex operation of changes (change propagation) which have to be controlled as well as otherwise the timing for the whole system could go astray. Therefore, tight watchdog mechanisms must remain optionally in place to guarantee that such update cycles do not cause trouble to overall system responsiveness as these are usually performed within one execution thread and thus cause stalling until performed.

When defining endpoints, it must be verified by the gateway as well that these endpoints do not form a cyclic dependency graph as otherwise the update would cause in indefinite recursion and break the process finally. Thus definitions of dependencies which are cyclic in nature have to be rejected or at least during runtime discovered and eliminated.

3.5.5 Communication interface agnostic

For an integrator it is always a big benefit if all operations are available in a similar way in all supported communication interfaces a solution offers. Given that, the integration focus can be put on finding the most appropriate communication protocol for the given task and not having to investigate if all operations are supported or might cause some undesired side-effect (for example throttling of requests if a request / sec rate exceeds some value in protocol A, which is not an issue for protocol B).

In addition, a wide support of communication choices is always beneficial. Currently there is a tendency to use only REST for most operations as it is so easy and quick to implement and extend, yet by the author's industry experience this can lead easily to incomplete designs and cluttered implementations that this in the long run can cause quite some problems and therefore more formal, and thus stable, protocols as SOAP are desirable as well. Having both options – and then all operations in both - is definitely a huge advantage as it allows to choose the optimal interface for the task at hand.

An exception to this rule would be dedicated protocols for dedicated tasks. So for example ODATA is designed primarily as a data request protocol for applications like Excel and thus the operations there (if supported by the gateway) are only valid in the specific implementation.

3.5.6 Autonomous operation / workflow execution

An autonomous operation is an action which is performed by the DBGA itself, based on rules for the execution (like for instance when a new value arrives), yet without the interference of an external actor to start something (so no user has to interact with the

system for the execution). The operation as such is implemented in a so-called workflow, where the workflow is usually a piece of code designed using BPML[80] and then generated at runtime, or in a more static implementation, yet still loaded and executed on-demand.

These workflows can be either short- or long-lasting, where long-lasting usually implies that they can be persisted while waiting for an external event and on arrival of that event (usually via some kind of correlation mechanism) will resume operation and short-lasting ones are meant to be executed in the same iteration they are called and used without interruption in terms of suspending and re-activation.

Autonomous operation (workflows) usage in a gateway comes into use in 2 situations:

- When based on the arrival of new values (as trigger) something should be done
- When cyclic tasks which have to run periodically are needed

In the first situation the usage is similar to the filter operations described in section 3.5.3 and the benefit by using a workflow in the very often more formal approach of definition as well as pre-defined infrastructure (for example actions) which are defined so that standard tasks can be performed easier.

The second usage depends very much on the application logic needs. Usually when complex background operations (like for example checking backend systems based on sensor data and using the result to generate new data) are needed, workflows are a good match. In case only some small computation is needed (for example running mean, etc.), a virtual value (see section 3.5.4) might be the better choice.

In general, workflows can be substituted by classical filters (usage one) or external applications which read sensor data periodically, perform the task to be done and write back values (usage two).

3.5.7 Actuator support

Writing to actuators can be achieved in several ways, depending very much on the environment a gateway is running in:

1) Directly write to the actuator from the gateway
2) Write to the actuator via an intermediary (usually a PLC)

---

[80] BPML = Business Process Modelling Language – a XML-based notation to model processes. This is usually

3) Provide a pool where messages for the actuator are stored so they can be retrieved by the actuator

Option 1 is usable in low-risk environments (for example not turning down a shutter in a house climate control system has potentially a lower impact than not closing a valve in a chemical process). This is due to the fact that execution can never be really controlled and especially not guaranteed, so it is not really reliable. Should this required, option 2 will be used.

In cases where this reliability is needed (or defined by policy like in the author's integration example – see section 7.2) then actuator integration is usually done using option 2 by writing the actuator value from the gateway to a PLC[81] using a for example private protocol or calling a web-service on the PLC. For the caller of the actuator writing the effect still is the same, the value will be written to the destination.

Option 3 is usually used by message-bus based systems which only know message producers and consumers. So the actuator would be the consumer who has to read messages from its inbox and process them afterwards. Usually in this case a so called TTL (Time To Live) is associated with the message which will erase it after expiration of the TTL so that old and outdated messages are not transported to the receiver anymore. This approach cannot be used for any integration which needs response time guarantees or even acknowledges of writing to the actuator as everything is asynchronous – here option 1 and 2 could provide the caller the guarantee that a write occurred.

3.5.8 Write target agnostic

When writing values to an actuator it must be possible that the actuator is either a physical or a virtual actuator. In terms of a physical actuator the address would be in any format which allows the gateway and the communication handlers to reach the actuator and submit the value.

For a virtual actuator the address would be URL where the gateway can just write to and the receiving end (usually a process) is responsible for further processing of the data.

---

[81] Which has all the necessary features like reliability, resilience, redundancy, etc. integrated to guarantee operations

This approach allows to easily integrate almost everything which can be written into a gateway environment as for example an alerting service which would be written to (see section 3.1.1 for an example) could be implemented as an actuator and thus any write to the actuator results in a write to the alerting system.

3.5.9 Data format agnostic

When data is stored internally in a format which is not specific to any CPU but in a human readable format a lot of problems like conversion, limited data types, etc. are eliminated. This way data is just an arbitrary amount of characters where the semantic meaning is left to the senders and receivers. The only problem which remains is that, for example, a floating number stored in one numbering format (for example using . as decimal separator) has to be converted back at the receiving client accordingly. The benefit of this approach is that any payload associated with sensor data can be stored which could be even XML or JSON data.

As size usually is no issue nowadays[82] this is no hindering either and the benefits of the uniform internal data handling (no different logic paths for different formats) pay off in cleaner and more straightforward code. Especially, as no semantic context has to be associated with sensor data any more – for the gateway it is just an object of data which has to be treated. When data is coming to or from the gateway using REST or SOAP it is anyhow converted into a textual format.

3.5.10 Preservation of sensor state

When a gateway starts up, either after initial start or after a fail, it could be – depending on the usage environment – very important for clients, that the last known sensor state is re-read from permanent storage and used automatically. This might mean that the value is outdated, but it is a value anyhow.

To handle this problem with outdated values several methods should be associated with the state preservation like a TTL (time to live), how long a value is considered valid, what to do after re-start (re-read from central database or assume an initial value), etc. Many things in the initial handling depend on the environment as well, yet quite a lot of scenarios can be handled by configuration and the gateway itself.

[82] storing an INT in perhaps 10 bytes instead of 4 would require billions of records to have a significant impact

3.5.11 Extendable and easy to change

As gateway projects are usually very dynamic and contain quite often new devices and new technologies to be integrated, an easy extension mechanism for a gateway is a very useful feature. This usually happens in all parts, so either a new device, a new communication protocol, a new filter, a new handler to achieve something…

Therefore, the architecture should be based on pluggable components so an extension / exchange of a feature / functionality can be easily achieved, and the integrator is able to react easily to changing requirements.

3.5.12 Provide feedback on data change

A common problem during integration is how a consumer gets the information that a data-point (sensor) of interest changed. Very often this is achieved either by polling, which causes additional load on the communication and server infrastructure, or, when using message-based systems, by placing a message in the inbox of the consumer which then would unblock any reader.

A much more straightforward approach is if the consumer is notified from the gateway if such a change happens and then can react accordingly. This notification could happen using a variety of mechanisms like Web-Sockets, callbacks, HTTP-calls, etc. which is again dependent on the integration environment. Yet the ability in general, when combined with a pluggable architecture, so that the mechanism is exchangeable, is very important for a reactive and fast solution.

3.5.13 Basic ontology-Support

Especially when integrating similar devices (from various vendors) often the problem arises that several sensors provide the same logical information yet are called differently. In the example presented in section 2.5 several chambers (manufactured by 4 different vendors) provide information which is temperature related. The device gateway could capture each sensor value and a consuming application access those values, yet every joining between different sensors across chambers (like for instance to compare the temperature curve in two chambers) would require the application to know which channel (sensor) provides temperature related data.

Therefore, with a basic ontology layer inside the device gateway the sensors could be grouped in the for example “temperature” group and any consumer could query the associated sensors and afterwards their values.

This way, by providing an arbitrary grouping of sensor attributes to form new query able attribute sets, the device gateway provides a very important meta-information layer for any consumer which makes data consumption much easier.

3.6 Comparison of requirements coverage by selected state of art architectures

In this section, the coverage of the derived requirements by the selected state of the art device gateway architectures are compared, including the proposed DBGA device gateway architecture. Due to their non-standard nature, custom/proprietary gateway solutions are not included.

Table 3.8 shows the requirements as rows and systems as columns, with individual cells including commentary about coverage (the most important ones – according to the rating in section 3.6. – have been ***indicated***).

Table 3.8: Requirement coverage for device gateway architectures including DBGA

#	Requirement	Device Business Gateway Architecture (DBGA)	Azure IoT	Xively	sMAP
R1	***Centralized data store***	Yes, easy to exchange by providing a new plug-in.	Yes, is highly configurable (SQL-DB, BLOB, flat files, etc.)	Yes (but not configurable and not options)	Principally per node, but could be configured to use a central repository
R2	Data retrieval using query facility	Limited	Yes	Limited	Limited
R3	***Direct data retrieval from data store***	Yes	Yes	No	Yes
R4	***Communication interface agnostic***	Yes	Not 100%, some functionality is not available in HTTP (for example throttling if messages are retrieved too often). Service endpoints only available using ACMP	Not 100%, some functionality is not available in HTTP	No, only REST
R5	***Data format agnostic***	Yes	Yes	Yes	No

#	Requirement	Device Business Gateway Architecture (DBGA)	Azure IoT	Xively	sMAP
R6	***Integrated data quality control***	Yes (filters)	Yes (has to be implemented)	No	No
R7	***Semantic data value enrichment***	Yes (virtual values)	Not integrated and online - only by means of writing triggers and then performing the calculations where the result is stored in secondary storage	No	No
R8	Autonomous operation / workflow execution	Yes (workflows can be launched based on data or in general)	Yes (workflows can be launched based on data or in general)	No	No
R9	***Actuator support***	Yes	Only indirectly by writing to an outbox	Only indirectly by writing to an outbox	No
R10	***Write-target agnostic***	Yes	Not directly, but possible with	No	No

#	Requirement	Device Business Gateway Architecture (DBGA)	Azure IoT	Xively	sMAP
			workaround to write to a new message-queue		
R11	Provide feedback on data change	Yes	Yes, needs special message processing	No	No
R12	Data Source Interaction agnostic	Yes	Only PUSH	Only PUSH	Yes
R13	Provide discovery support for devices / sensors	No	Yes	Partially	No
R14	Authenticate users and limit access	Limited	Yes	Partially	Yes
R15	***Preservation of sensor state***	Yes	No (messages for the sensor have to be read to identify the “last” valid value)	Yes	Yes
R16	Ontology support		No	Partially	Partially
R17	Architecture should be implementable using "free" software	Yes (yet Windows is required, Mono is an option)	No	No	Yes (running on LINUX)

#	Requirement	Device Business Gateway Architecture (DBGA)	Azure IoT	Xively	sMAP
R18	Extendable and easy to adapt to changing requirements	Yes	Principally, but rather complex and steep learning curve is involved. Many different options and configuration points as IoT is not a single package but a combination of many different parts	No	Moderately as protocol support is limited, yet in the given operational limits it is quite easy to use
R19	Based on modern software engineering practice	Yes	Mostly recycled older technology like BizTalk which was extended and enriched with some new features. The API and programming	Yes	Principally, yet a bit aged software (already 10 years)

#	Requirement	Device Business Gateway Architecture (DBGA)	Azure IoT	Xively	sMAP
			interface as well as the internal messaging engine is new technology (.NET)		
R20	Operations support	Partially	N/A (hosted solution)	N/A (hosted solution)	No
R21	Integration of a new device should be straightforward	Yes	Yes (as long as the protocol stack is accessible from the device)	Yes (as long as the protocol stack is accessible from the device)	Yes
R22	Provide scalable and fault tolerant infrastructure	Can be done (for example SQL clustering, hot standby and failover)	N/A (hosted solution)	N/A (hosted solution)	No

As the DBGA provides a flexible architecture to store data centrally (which could be in a NoSQL-database as well[83]) as well as a direct data retrieval from the data source data operation requirements are fulfilled.

By providing an agnostic communications interface the DBGA can be accessed from any supported client with the same functionality regardless of the communication method which is a major difference to the compared solutions (where this can be a limiting factor in actual implementations).

An integrated data quality control is a major difference to the other solutions as either it is not possible at all, or has to be implemented in quite difficult and time-consuming ways.

By providing actuator support, which is a mandatory feature in any industry automation environment which has to control something (as otherwise it is a read-only operation), the DBGA offers a very different approach to the integrator. This goes hand-in-hand with write-target agnosticism to be able to write to any kind of destination (so either physical or logical device which could be another process).

Yet perhaps the two most important requirements for a typical industry automation environment are the preservation of sensor state and the ability to have a semantic data value enrichment. State preservation is needed to relieve the client from tracking sensor states itself which makes Azure IoT (and most message-based systems) quite hard to use from an integration point of view as these states have to be retrieved upfront (to know where a system is starting from) and cannot be queried.

Given the discussion about value-chain-management and integration of cyber-physical-devices (sensors, etc.) into value-chains ([84], [87]) data value enrichment in Azure IoT is quite difficult to implement (yet can be done very flexibly given enough work is invested), whereas for the other solutions there is no direct option available.

So, it can be summarized that the DBGA covers most requirements and especially those which have been rated in section 3.6 as important. The ones not provided in the current situation can be added in a later stage and are not necessarily needed for an industry automation use-case.

[83] Which might make sense if there are very big amounts of data records

Having defined the requirements for and the characteristics of the DBGA, the next chapter focuses on how these can be converted into a software design (chapter 3.7), which is then used for a reference implementation in chapter 5.

3.7 Measuring the improvement achieved by the architecture

Concerning the research question the measurement of the improvement of the business process integration is of paramount importance as only by these indicators will any comparison among architectures be possible. The problem is that no such criteria exists on a general basis. Everybody doing integration focuses and covers different points, which the author of this thesis has experienced during 32 years of working experience in the industry automation area as software developer and solution architect.

Some like [75], [76] or [29] use the performance difference of the overall process, whereas others like [74] or [77] are more interested in the net results this integration provides (less waste, less recalls, less field-visits, reputation increase, and so on). Therefore, the criteria selected and used in this thesis are based on the significant personal experience of the author of this thesis in business process integration projects and reflects more on improvement upon the direct integration part itself, and not so much the improved "gain" or "benefits" upon the parts using the integration.

To measure the process integration improvement, several criteria which are shown in Table 3.9, were decided upon by the author of this thesis as reasonable and confirmed by a key business stakeholder in the automation industry[84].

Each criterion is defined by describing the measure that will be applied during the analysis, and the meaning of the criteria. The Level of achieving the requirements ranges from 1 = low achievement to 3 = everything (over)achieved and (subjectively) measures how the requirements by the use cases were achieved by the presented solution (or not).

[84] These criteria were confirmed as reasonable Mr. Robert Mayrhofer, Head of Corrosion Testing Team (KPZ), Daimler AG

Table 3.9: Measurement criteria definition to compare different architectures

#	Criteria	Levels	Meaning
M1	Level of achieving the requirements	1 = low achievement 3 = high achievement	A general indicator expressing how many requirements have been achieved
M2	Flexibility offered by the architecture for an integrator to integrate devices into business processes	1 = very inflexible 3 = highly flexible	A general indicator expressing how easy a device and sensor can be integrated into a business process. This is a very general measure as it has to be reflected upon using the concrete use case at hand. Therefore, for this evaluation parameters like communication protocols provided, agnosticism in protocols, ease on understanding, etc. were considered
M3	Performance	1 = low performance, 3 = very good performance	Estimates the performance the solution will provide and anticipates how it will behave with the "real" workload
M4	Maximum number of devices / sensors / actuators supported	Any number or a range	Are there any practical / theoretical limits to the number of devices / sensors / actuators?
M5	Maximum number of sensors which can be handled as peak	Any number or a range	How many sensor value changes could be handled in a given period of time

#	Criteria	Levels	Meaning
M6	Complexity	1 = low complexity, 3 = highly complex	The level of interdependence between involved subsystem [59][85] where 1 would mean little interdependence and 3 high inter-dependences. A typical example of 1 would be that parts can be exchanged easily and directly without affecting the overall system whereas in 3 an exchange must be meticulously evaluated to avoid disaster. Very often seemingly non-complex systems afterwards prove to be highly complex as the interdependencies had been very high or non-standardized
M7	Skill Required	1 = low skill requirements, 3 = high requirements	Defines how much skill is required from an implementer to create / realize the solution. This includes actual skills in customization / programming as well as general skills for the environment
M8	Learning Curve	1 = low curve, 3 = high curve	How much learning is involved to get the solution done if no previous knowledge (aside core essential knowledge) would exist
M9	Time required building	in hours	Here a rough estimate in hours divided into the following areas is given:

[85] Despite the fact that complexity depends on the observer

#	Criteria	Levels	Meaning
			• Conceptualization • missing skill acquisition (which always is the case) • realization • testing • fixing, adjusting For use case 2 the time for the app on the smart phone is not included
M10	Time required operating and administrating in one year	in hours	A rough estimate of how much time in h p.a. is needed to operate and administer the solution
M11	Ease of change in future	1 = very easy to change, 3 = not easy to change	Defines how easy it is to change a solution and to adapt it to new requirements. This combines the simplicity of design to some extent with the complexity and the skill required. If a simple design has a huge complexity and requires highly skilled people the ease of change is low (1) whereas if for example a very complex system has only moderate demands towards skills and a

#	Criteria	Levels	Meaning
			low learning curve as the components are clearly identifiable, then the ease of change would be high (3)[86]
M12	Preservation of investment	1 = low preservation, 3 = very high preservation	Evaluates how the solutions will preserve the initial investment over time (especially looking towards the point when changes are needed). This is a combination of the ability to change in the future Very often very simple (and inexpensive) solutions cannot be extended as very high skilled labor is needed which might result in a low preservation factor as the follow-up costs can easily outnumber the initial costs
M13	Free software / commercial software - cost		If software used in the solution is not free a possible cost in EUR is given
M14	Security	1 = not secure, 3 = highly secure (or can be made so)	Investigates the security the solution can (or could using all standard methods) offer in communication, data storage and invalid use (URL script injection, etc.)

[86] In reality all mixed forms would occur as well so it is really more a rule of thumb

#	Criteria	Levels	Meaning
M15	Reliability / Fault tolerance	1 = not reliable / fault tolerant, 3 = highly available	Defines if the solution is (or can be, using standard mechanisms, made) highly available and reliable so that 99.9% availability[87] can be achieved (minimum requirement for 24/7 operation[88])
M16	Overall stability	1 = highly stable 3 = can be volatile	Defines a global factor considering how stable the entire system is when: • devices fail (for sending / receiving) • parts of the infrastructure fail (for example database systems) Included is an observation about what could be done as well
M17	Easiness to obtain the current device / sensor state and use it in the business process	1 = very hard 3 = very easy	As the current state of a sensor is very often needed for business process decision making (for example if a value is exceeding a certain threshold) this measure shows how easy it is to obtain and use the current state
M18	Footprint of the system	1 = low footprint 3 = large footprint	How much infrastructure and overhead in terms of “environment” is needed to run and operate the architecture

[87] Which allows for 8:45h per year of non-availability

[88] Usually the request is made to have 99.99% as a goal (52 min downtime / year), yet the costs usually will rise exponentially, so that normally 99,9% is a practicable value

4 Device-Business-Gateway Architecture (DBGA) design

The device gateway is a very typical example of a service-oriented architecture where application components provide services to other components via a communications protocol, typically over a network, and are independent of any vendor, product or technology.

In this thesis, the gateway is a service which is offered and consumed by all sorts of clients, which acts as the central mediator between data senders and consumers, performing processing and being accessed by visualization interfaces.

On a very general and abstract architectural level the following statements of principle underlie the design:

- As the gateway has a universal approach (in not being limited to any specific kind of client) serving devices, sensors and very technical as well as business process clients, a multitude of stakeholders with different concerns have to be satisfied. Therefore, the design has to reflect this multidisciplinary nature by offering services and communication facilities suitable for each stakeholder.
- Device data usually is extremely crucial for ongoing operations, thus quality attributes like extensibility, reliability, usability and other such "-ilities" (all non-functional requirements) are in close relation to the architecture. In accordance with current research [64] the architecture proposed in this thesis is mainly driven by the stakeholder's concerns regarding these attributes and not so much by classic software design approaches (for example Jackson Structured Programming) where required functionality and the flow of data through the system are the main issues. As a result, the architecture prioritizes these quality attributes over feature / functionality.
- The design is entirely separated from the implementation and provides the overall vision of the whole system. It dictates what and how the system should act without being limited by programming language or environmental issues of an implementation, thus providing conceptual integrity. The reference implementation's goal is simply to provide a test-bed for the necessary requirements tests and to build the evaluation scenarios. No impact from the implementation must influence the design.
- The device gateway is designed in a runtime agnostic form, which means that its later deployed runtime environment has no impact on the design as such. By

using abstraction layers whenever possible and needed, the necessary encapsulation of features against the environment can be resolved.

- In case specific options are available and usable[89] the most suitable ones for a broad and encompassing solution are chosen. In general, the selected options represent an approach based on the current situation and could easily change in the future (which is the reason why for example Ruby is no longer supported as a virtual value evaluation language).
- Design patterns (see [67]) and service design patterns (see [66]) in particular are used wherever possible in the design as a provision against repetitive definitions. To further enhance the design several Anti-patterns[90] (see [68]) were considered, too.
- To allow for the best separation of concerns the view model approach, as designed by Phillipe Kruchten [63], is used. UML is used as the notation in the design. Contents of the view model approach include:
    - Section 3.1 describes the scenarios and the use cases are described in section 3.2, and these support the gateway architecture design (scenario view) which serve as a basis for the case studies in Chapter 7;
    - Section 4.3 will describe the design of the components of the gateway architecture design (logical view);
    - Section 4.5 represents the dynamic aspects of the interactions of the components in the gateway architecture design (process view);
    - Section 4.6 depicts the topology of components which form the gateway architecture (physical view).
- In addition to these, section 4.7 presents the data model as an architectural view [70].

The support for various data sources and data destinations is described (section 4.1), followed by the design of communication (section 4.2), a description of the workflow usage (section 4.3) and the logical view of the components (section 4.4), as well as the dynamic view in section 4.5. The physical view is described in section 4.6 and the data model in section 4.7. The chapter is concluded by section 4.8.

---

[89] For example, virtual value evaluation in section 4.5.7.4 can be done in a variety of ways using either JavaScript, Python, or any other dynamic resolvable language environment

[90] Typical examples might be the use of patterns everywhere (even if a non-pattern approach might be better suited) or selecting ill-suited patterns for a problem

4.1 Data sources and destinations (data entities)

Data sources and destinations (also called data entities) and their value(s) are the core data structure defined and handled by the gateway as everything revolves around them. Therefore, the core definition of what data sources and destinations are, as well as how they can be chained is of paramount importance to the whole design of the architecture.

A data source or destination of the gateway is a uniquely addressable entity (by whatever protocol and definition) which represents a value. In the case where an entity does not exist in real life, but only in the DBGA as a conceptual value, it is called a virtual value. Therefore, a data source or destination could be actually anything from a real physical sensor or actuator to a virtual sensor or actuator representing a dynamically evaluated value.

Devices can either exist as physical or virtual devices. A physical device usually implies a tangible "thing" (the device) which can be "communicated with" by means of some communication media and protocol. Virtual devices exist just as a definition for an abstract data set [of data entities], where the sensors or actuator (which could be literally unlimited) are the single data items (endpoints). As an example, for the gateway a REST-based web-service could be described as a virtual device, having the resources (which can be addressed by REST) as the sensors and any reading would yield the current sensor value of the service / resource combination which is represented by that concrete endpoint. Therefore, virtual devices and sensors are treated equally to physical devices in every aspect.

Generally, it could be defined that: the gateway considers a data entity, which has readable / writable values and can be classified in terms of device / sensor, as an endpoint usable for further operation.

Data entities can be read, write or read-write, depending on their use-case and supported protocols. The gateway has to enforce that a write only entity will not be read and a read-only not be written to.

4.1.1 Registration of data entities

The registration of devices and data endpoints (either sensors or actuators) must be done explicitly by means of the provided API of the device gateway. This is necessary for any kind of data entity including virtual "non-store" endpoints, which are just computed on the fly and have no persisted information whatsoever.

It is during the registration where all definitions are made, like what kind of endpoint (entity) the data is - either sensor, actuator or both, whether it is a virtual value and how the dependencies are, and so on. So, all the vital basic information about an item is given here, which can be changed later, of course.

During the registration process, when a data entity is registered by the gateway, housekeeping tasks have to create the necessary data structures, and so on.

4.1.2 Virtual devices / virtual sensors entities

By using the same analogy as physical devices, a logical device acts as the container for several virtual sensors and thus provides 1..n endpoints as entities which can be addressed like physical endpoints. What is behind or underneath such an endpoint is in principle of no concern for the consuming application as it behaves exactly like a physical endpoint would do.

Classical examples for such virtual devices / sensor combinations would be (not conclusive):

- A “Statistics” virtual device with sensor values for the different statistical methods like Min(), Max(), Mean(), etc. for physical sensor readings;
- An “External data” virtual device with sensor readings for energy costs / kWh, etc. from a provider’s web service;
- In addition to these more "classic" examples a virtual entity could be any value evaluable by a process like the memory consumption, CPU utilization, requests made, and so on.

By using these virtual entities data handled by the gateway will become semantically enriched as not only actual physical data, but new data, based on definitions, calculations and relations can be generated. In this way existing data can be combined to form new data, thus allowing additional insights and especially move the process of these calculations from the consumer of the data to the source (the gateway) making it therefore much more (quality) controllable, (centrally) adjustable, available and valuable for everybody.

The actions involved behind getting / setting a virtual entity are described in section 4.5.7.

These virtual entities act logically in very much the same way as a user defined function in a SQL database does – providing a server-side data item which can be queried without the need of the client to either fully understand the underlying principles or

connections behind the data. In addition to this analogy virtual entities can be written to, as well.

So, in principle any internal or external data item could be masqueraded behind such a virtual entity definition. In this way it is the task of the virtual device and not relevant for the consumer how the device gets the sensor readings (or in case of actuators writes back values). It is just important that necessary overall timing, stability and security issues are preserved.

4.1.3 Data entity trees

Data entities can form trees or chains (as a special form a linear tree with no branches) of entities. In such a tree root nodes, which rely upon other nodes' values, have to be virtual entities whereas leaf or base nodes (one node where other nodes rely upon) can be either a physical or virtual entity.

Currently such entity trees can only be defined for reading of virtual values, as writing involves many specific problems and is a deferred problem. Saying this it is possible to write a virtual value, of course - just no further propagation of such a write in a tree occurs.

Every node in the tree can be queried individually as an independent value, yet the higher up in the hierarchy the query is addressed, the larger the evaluation basis becomes as more base nodes have to be evaluated. Using this approach, a step-by-step aggregation of information can be achieved as each step could be represented by an independent virtual value, where others are based upon. This is very similar to a function calling other functions, and so on, in a programming language.

Normally virtual values are evaluated "on demand" by default, which means that the evaluation is performed when the value is requested by a consumer. As this evaluation can be a quite time-consuming operation and perhaps involve querying sub-sequent physical entities (which involve further potential time-delays) the gateway design provides the option to automatically update dependent values. Here a change in a base value (which other entities are dependent upon) is directly propagated up the tree and the new dependent value is evaluated at the moment the change takes place. This can be either the response to a newly read value or a written one and works for physical as well as virtual entities.

A typical example of such an entity tree is shown in Figure 4.1. Two physical values are presented that both automatically propagate any change up to their dependent value. This implies that the value for "KPI 2" is always up to date, as both dependencies are

always synchronised with the value. This is not true for "KPI 1" as this only receives updates from "Physical Value 1", but not "Business Process Value 1". Therefore, the gateway engine cannot automatically serve the current value when queried (like possible for "KPI 2") but has to start the virtual value evaluation process. Here it could be implemented that anyhow the actual value is taken as for example "Business Process Value 1" is more like a constant, but this is entirely up to the implementation of the virtual value evaluation.

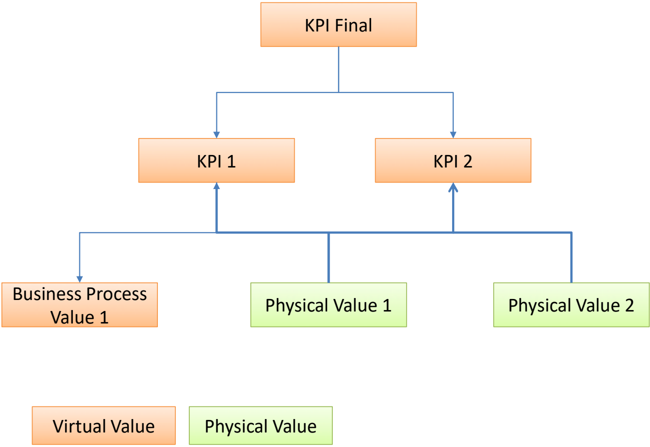

Figure 4.1: Data Entity Tree

Querying "KPI Final" or "KPI 1" will always involve computing the virtual value as their values have to be computed when queried.

4.1.4 Virtual entity value evaluation (computation)

To ease the business process integration process where virtual value evaluation is a very important task, the gateway design allows for great flexibility in the way the actual evaluation of the value of a virtual entity is performed.

In general, the evaluation module will be passed the current environment (as access point to the device gateway) as well as the virtual entity to be evaluated and is expected to return the new value. Using the environment, the module has access to all methodology of the gateway including access to all other values defined in the system.

As virtual value evaluation is a non-predictable process due to code provided by external sources, the design has to take precautions against system instability. The following mechanisms to secure system integrity are provided:

- Quota assignments which mean that only x number of requests are allowed within a given timespan;
- Constant monitoring of the CPU utilization and if given factors are exceeded further evaluations are gracefully degraded;
- Execution of external code is always and only done inside an independent unit of execution, so that a failure (for example exception) inside a module (for example division by zero) causes no threat to the overall system.

As semantic data value enrichment (section 3.5.4) is a major characteristic which requires virtual value evaluation to function, an implementation of an architecture should provide several options to allow the integrator to choose the most appropriate one. For implementation details of the reference implementation for this matter refer to section 5.4.1.2.

4.1.5 Performance considerations of virtual value evaluation

The performance of the virtual value evaluation cannot be defined or predicted as it depends entirely on the actual implementation of the evaluation (in terms of for example JavaScript or Python modules). Even more important is the approach taken in this thesis, to provide as much support for performance optimization as possible and therefore easing the integrators burden to care for these topics.

Despite all parallelism and distribution of work inside the gateway, for the consumer the total request duration between issuing the request for a virtual value and the receipt of the result is still the only relevant timing. This duration has to be optimized as much as possible wherever it can be influenced by the architecture.

One option, to enhance request performance (total duration) is that the design defines thresholds until when "current" data is still to be considered "current", which eliminates the need to constantly re-evaluate still valid data. So for example in the case of a slowly changing base value the threshold for a virtual entity could be 2 hours, meaning that any request for that virtual entity within 2 hours since the last evaluation will be served the current value.

Due to the importance of virtual value evaluation, several test cases have been designed especially for the evaluation of timing, load factors and corresponding limits in section 6.3 and 6.4.

4.2 DBGA communication

Communication with the gateway is a task undertaken by consumers as well as producers of data (like sensors). It is very often the case that consumers (more details

about consumers can be found in section D.2 (appendix)) choose different protocols than those of producers which very often dictated by business integration needs, skills and more technical factors like latency, throughput, and so on.

The gateway design uses the same communication handlers for inbound (receiving) as well as outbound (sending) communication. So, when writing from the gateway (by means of an actuator value being written to an endpoint), the same handlers as for reading are involved.

4.2.1 Communication interface agnosticism

As the architecture should be communication interface agnostic (see section 3.5.5) several communication interfaces and protocols like SOAP, REST, ODATA, CNDEP and MS-MQ[91] (among others as an extendable list) have to be supported.

A client can communicate with the DBGA in any defined way and it has to be irrelevant over which interface (even mixed) the communication is undertaken. Here the communication interface exposes the same underlying logic and API, yet in a communication interface specific format to enable the use of interface specific characteristics. In this way the DBGA "looks" different on all communication interfaces, yet acts the same. In case a communication interface does not support all operations (for example ODATA is usually used to read bulk data, but not to write new values), the communication handler has to signal according result codes / exceptions to the user.

Further details about the various protocol specific issues can be found in section D.3 (appendix).

4.2.2 Device specific communication

Between receiving and sending data there is one big difference which has to be considered in the design.

When the gateway is the receiver, the gateway defines the logical protocol (API) to be used with variations for each underlying communication protocol. When the gateway is the caller (for example by means of scanning data endpoints or writing to actuators) the gateway has to obey the logical protocol definition defined by the called endpoint. This means that for example scanner tasks (for details see section 4.5.4) can use the

---

[91] MS-MQ = Microsoft Message Queue -> Microsoft's product (as part of the Windows operating system) to provide reliable message queuing (https://msdn.microsoft.com/en-us/library/ms711472(v=vs.85).aspx)

generic communication interfaces (interface agnosticism), but need specialized logic to handle the logical protocol with the data endpoint. To cope with this implication each data endpoint that is considered to be pulled (scanned) has to provide the class which is capable of communicating with the endpoint. It is the sole responsibility of the class to call the necessary APIs of the endpoint and thus provide a seamless and generic integration into the device gateway framework

4.3 Usage of workflow execution as autonomous operations in the DBGA

To implement the characteristic of autonomous operations (specified in section 3.5.6) the DBGA design uses workflows to achieve this.

The definition for what a workflow is tends to be blurred at best, so for the design the following definition, is assumed[92] (see [72] as well): "A workflow is the definition of a shared and usually repetitive business process. Through the workflow the tasks, processing units, the structure and inter-operation between participants as well as data-flow and process-logic-flow are defined”.

These workflows would be usually designed in an external application and interact with the DBGA using a defined set of methods and a specific environment. They are either created on the fly from their BPML definition or loaded as runnable modules (if present in an executable state) into the DBGA runtime and then executed on demand.

Based on this definition the device gateway incorporates workflows in two areas:

- As call-backs to for example calculate a virtual value or react to a change in a sensor value;
- As cyclic tasks which are executed automatically in the system.

Whereas the second use case exploits all characteristic of a workflow to its full extent (especially the autonomous operations and proactivity), the first use case degrades the workflow more to an execution engine for an action, as a reaction to an external event which was sensed by the device gateway. In any case, the workflow can do whatever it has been developed to do, mainly setting other values (actuators) based on the current state of its inbound sensors.

[92] https://wirtschaftslexikon.gabler.de/definition/workflow-48807

In the .NET environment a workflow would be implemented as a Windows Workflow Foundation[93] workflow which is run and controlled inside the Device Gateway.

4.4 The components of the device gateway (logical view)

The device gateway architecture is composed of components that resemble the standard use cases (see section 3.2). Table 4.1 shows a mapping between these use cases and the component(s) involved.

Table 4.1: Use-Case to Component Mapping

Use Case	Involved Component Name(s)
Manage devices and sensors	• Device and Sensor Management (section 4.5.1)
Write sensor value	• Value Management (section 4.5.7) • Receiver Tasks for Data Endpoint Issued Writes (section 4.5.3) • Sensor Scanning Task (section 4.5.4)
Read sensor value	• Sensor Read (section 4.5.5) • Value Management (section 4.5.7)
Evaluate virtual sensor value	• Value Management (section 4.5.7) • Virtual Value Evaluation (section 4.5.7.4)
Write actuator value	• Actuator write (section 4.5.2) • Value Management (section 4.5.7)
Validate value	• Value Management (section 4.5.7)
Cyclic execution of tasks	• *Watchdog and Cyclic Execution* (section 4.5.7.5)
Data access	• Data Access (section 4.5.6)
Access Control	• Access control (section 4.5.9)

These components are shown in the UML component diagram in Figure 4.2 where the relation with each other is described.

[93] https://msdn.microsoft.com/en-us/vstudio/jj684582.aspx

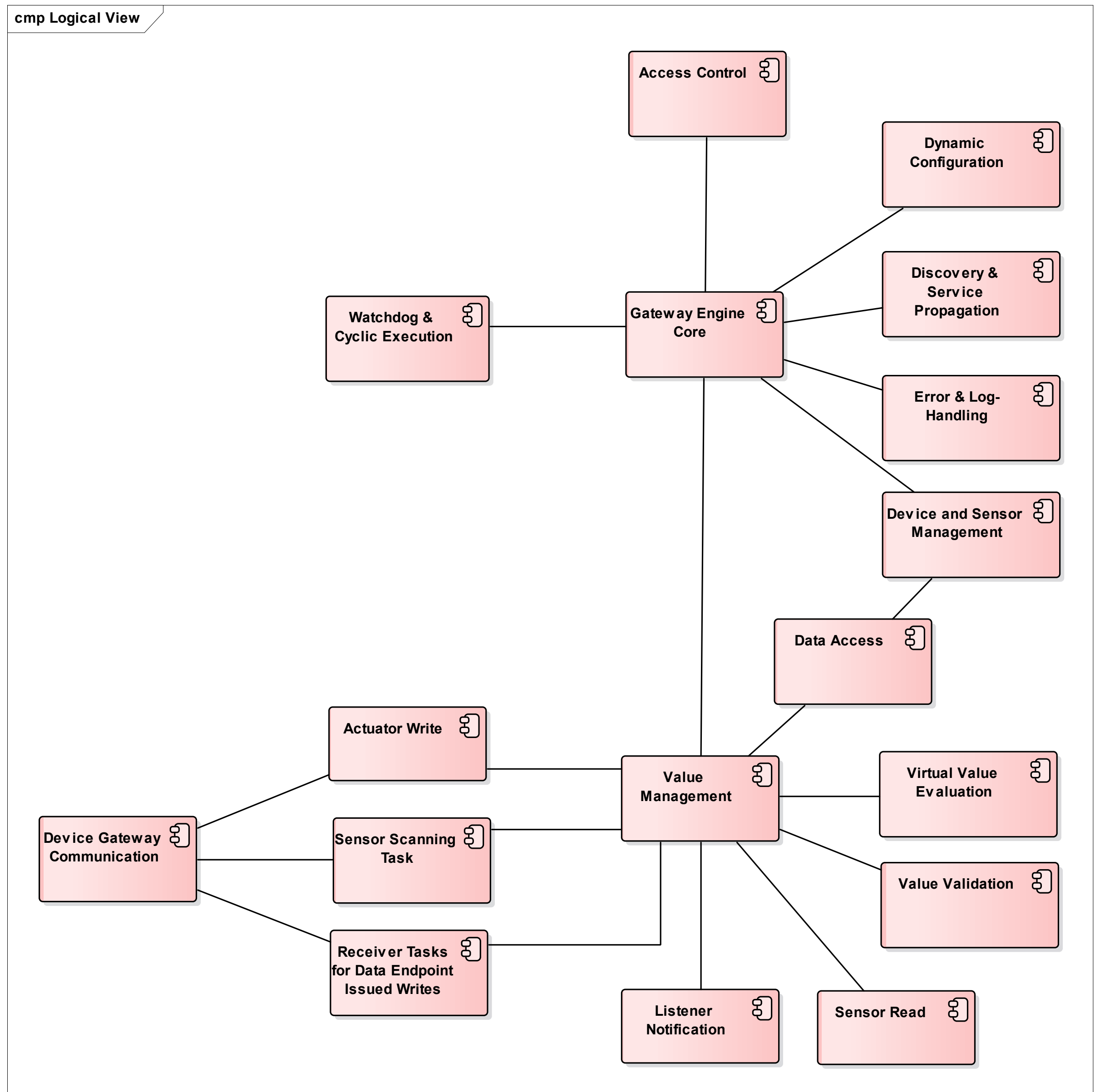

Figure 4.2: UML Component Interaction

Additional utility components like error and log handling, configuration, etc. can be found in the appendix (section D.4.3 to D.4.5).

4.5 Process (dynamic) view

To define the architecture's dynamic aspects, system processes, components involved and the internal communication between them, the usage scenarios (see section 3.1) will now be detailed using process views.

4.5.1 Device and sensor management

All devices and their associated sensors (and / or actuators) are managed by this component. It is mainly designed as a centralized registration place so that new devices / sensors / actuators can be easily registered and retrieved. Using an optional (not

designed and included) query facility this information can be queried as well. This way a discovery mechanism could be added later on as well.

4.5.2 Actuator writes

When writing actuator values the same problems as with scanning for endpoints occur as each actuator defines its own API and communication protocol to use which has to be obeyed. Therefore, the same rules apply for both usage scenarios so that actuator specific classes have to be present, which have to implement the *IDeviceCommunicationHandler*[94] interface.

An additional requirement for actuators exists that (depending upon the actuator and use case) some actuator data must be written synchronously at the exact moment when it is generated (when for example used during a workflow processing step), and some data can be written asynchronously. In general, the goal should be that data is written asynchronously as often as possible, as synchronous operation interrupt the normal "flow" of the system. This is exaggerated when for example a workflow is run (synchronously) based on a change of an input value and then sets an actuator value, which is written, again synchronously. This way the whole input handling is delayed (due to the synchronous following operations) until the final writing (with all potential time-outs, errors, etc.) has been handled.

To handle these requirements one potential design would have been to use a similar architecture like the scanner tasks, just for sending. Yet as writing data to actuators usually (according to current experience) happens not as often as receiving or reading from endpoints a simpler design was used for the device gateway. In this approach, shown in Figure 4.3, the task for writing resides (shielded from the remaining parts of the core) inside the device gateway core as an independent thread of execution which is launched during start-up. Should need arise this thread could be removed from the in-process execution and put into an external process.

[94] Tagging them as providers of the device communication classes

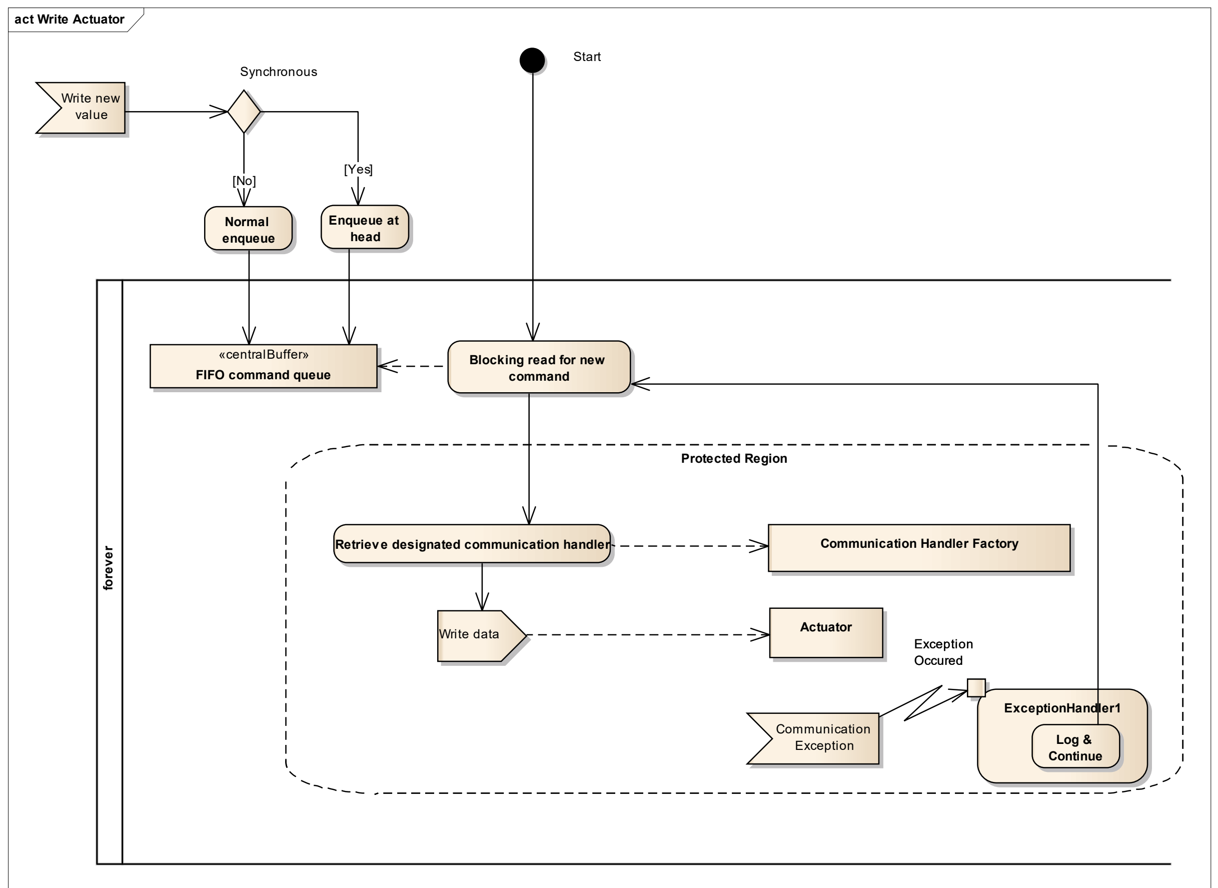

Figure 4.3: Writing actuator value

4.5.3 Receiver tasks for data endpoint issued writes

As the device gateway defines the API when data endpoints write their values to the gateway it is a mere matter of handling the various protocols over which the requests can arrive. Figure 4.4 provides a UML component diagram of these receiver tasks and data endpoints.

The protocol handlers are mere protocol converters between external representation and internal format. Thus, the load is not very high and for the predominant use-cases (REST, SOAP and binary formats) the system infrastructure will take good care for efficient parsing and routing of requests.

Further details about the specific protocol handlers can be found in section D.2 (appendix).

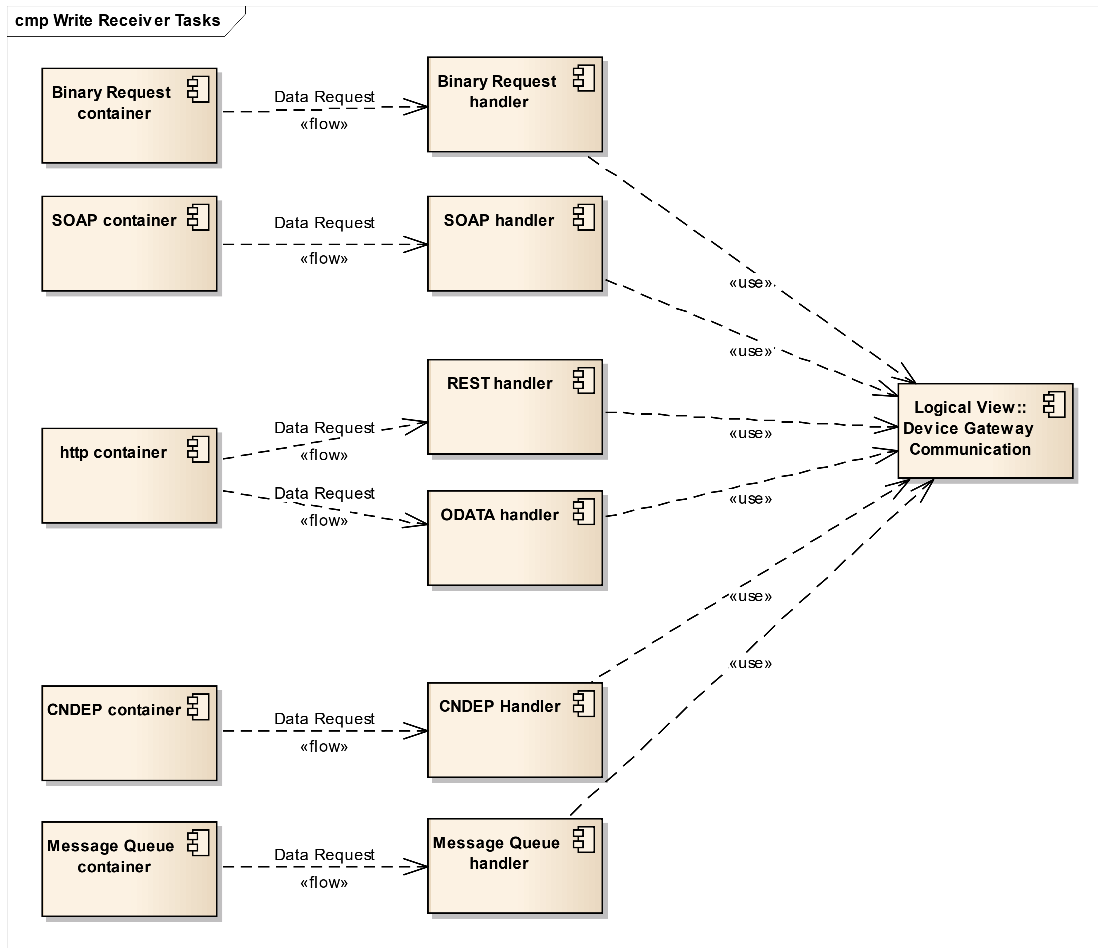

Figure 4.4: Receiver tasks for endpoint write requests

4.5.4 Sensor scanning task

To read values from devices and sensors attached to them is one of the main tasks the gateway has to perform if the sensors do not send their own values as they change. This scanning (pulling) of data endpoints has to be performed in so called "sensor scanning tasks". As each data endpoint can dictate its own API and communication protocol, the specific access has to be provided in a dedicated object (class instance) for this endpoint. The class to use for the communication is registered with the endpoint and used during runtime to create the specific class.

To enable best system performance and throughput several scanner tasks can be run at the same time where each of them is controlled by a central scanner task controller. This controller, by running in its own separate space (and not as part of the device gateway) is protecting the entire system against errors and exceptions arriving from the communication layer. In case of a serious problem "only" the task controller has to be restarted and the rest of the system remains responsive.

The ability to have several scanner tasks cope with situations when big latencies are to be expected and then the scanning of all data endpoints, would not be possible in a given period of time. A typical case might be querying devices using the SOAP protocol where, due to name resolution, XML parsing, etc., the response could be substantially delayed. If there are several endpoints to be queried using this approach the latencies, then add up and the total iteration takes (perhaps too) much time. Another use case might be when very different scanning times are present and it would thus make much more sense to group the scans into different scanner tasks.

The interactions between the task controller, the scanner task and the sensors are shown in Figure 4.5 as a UML diagram.

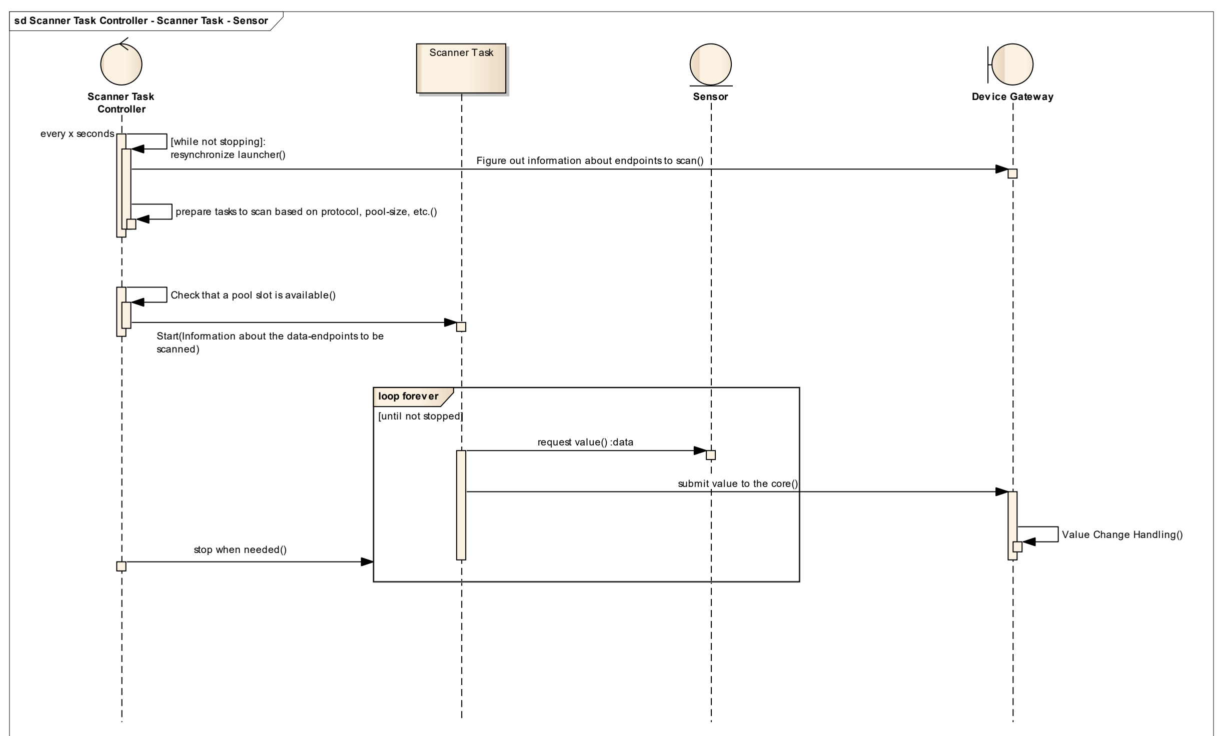

Figure 4.5: Task Launcher - Scanner Task - Sensor

It is the responsibility of the task controller to control how many scanner tasks are needed, when they are launched, when dismissed (destroyed), etc. As each task is considered an independent unit of work, the tasks are designed to be run in threads taken out of a (configurable) thread-pool so that no inflation of threads happens, which would degrade system performance again[95] due to synchronization, race-conditions and especially internal locks over shared resources. With the given thread-pool, system

[95] Some additional background can be found here: http://www.drdobbs.com/tools/avoiding-classic-threading-problems/231000499

behaviour can be fine-tuned towards the actual implementation scenario so that exactly the right amount of threads is used.

The scanner task itself is designed to run forever in an endless loop where parameters like scanning frequency, protocol, etc. are passed during creation. In this loop it then scans the sensors assigned and reports read values to the device gateway core where the value change handling (see section 4.5.7.1) begins. When the task controller needs to stop the scanner task a stop command is issued. Communication between scanner task and task controller has to be implemented by a FIFO-queue so that both sides can continue to operate and have a de-coupled media to exchange information.

These operations inside the scanner task are shown as a UML diagram in Figure 4.6:

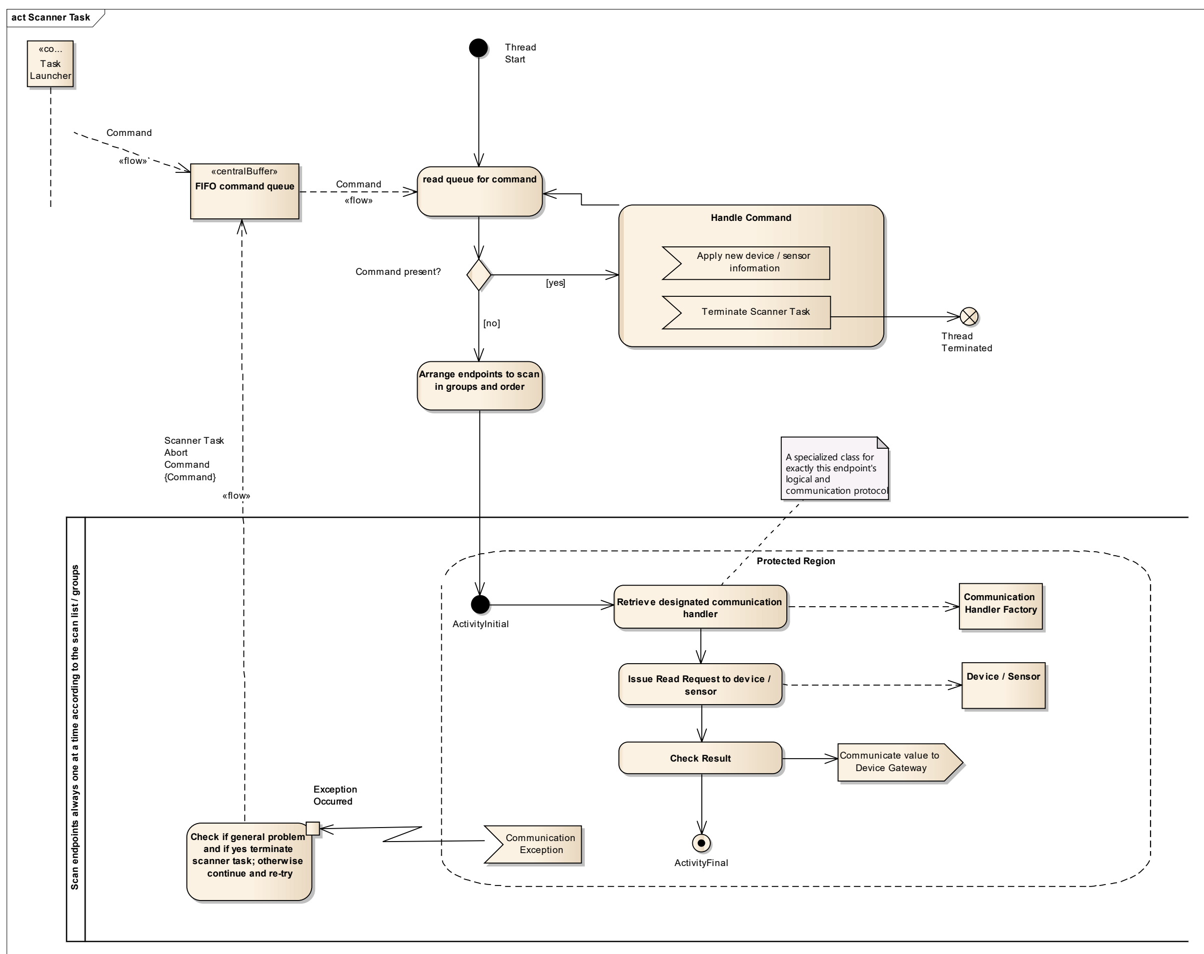

Figure 4.6: Operations inside scanner task

4.5.5 Sensor read

When a consumer wants to retrieve from the device gateway, he will issue a request towards any of the defined entry-points, which are 100% similar to the ones specified in "Receiver Tasks for Data Endpoint Issued Writes" (section 4.5.3). There the handlers are expecting sensor write operations; whereas the very same handlers, when used to

retrieve data, just receive the corresponding requests for retrieve operations. As the device gateway has a complete communication interface agnosticism characteristic (see section 3.5.5) it does not matter which communication protocol is used, as long as the protocol specific issues are addressed properly.

The basic parameters used for retrieving data are always:

- Data endpoint (sensor) relevant
- Optionally from when data is relevant
- Optionally until when data is relevant
- Maximum amount of results

These are then packaged according to whatever format is used during communication and could take forms like (dynamic parts are enclosed with {}) for example in REST:

- **http(s)://server/**SingleDevice/{DeviceId}/SingleSensor/{SensorId}?generatedAfter={Timestamp}&generatedBefore={Timestamp}

Usually to make requests shorter (and easier to use / learn) there are shortcut-versions available. For example, to get just the last value this would be:

- **http(s)://server/**SingleDevice/{DeviceId}/SingleSensor/{SensorId}/latest

All the requests are processed in the corresponding handlers and passed down, using the internal communication protocol to the device gateway kernel. There the request is further analysed and data retrieved using a global component *ValueManagement* (see later) which manages all values, caches, etc. in the system.

This is shown as an UML sequence diagram in Figure 4.7 where the communication between the components is represented.

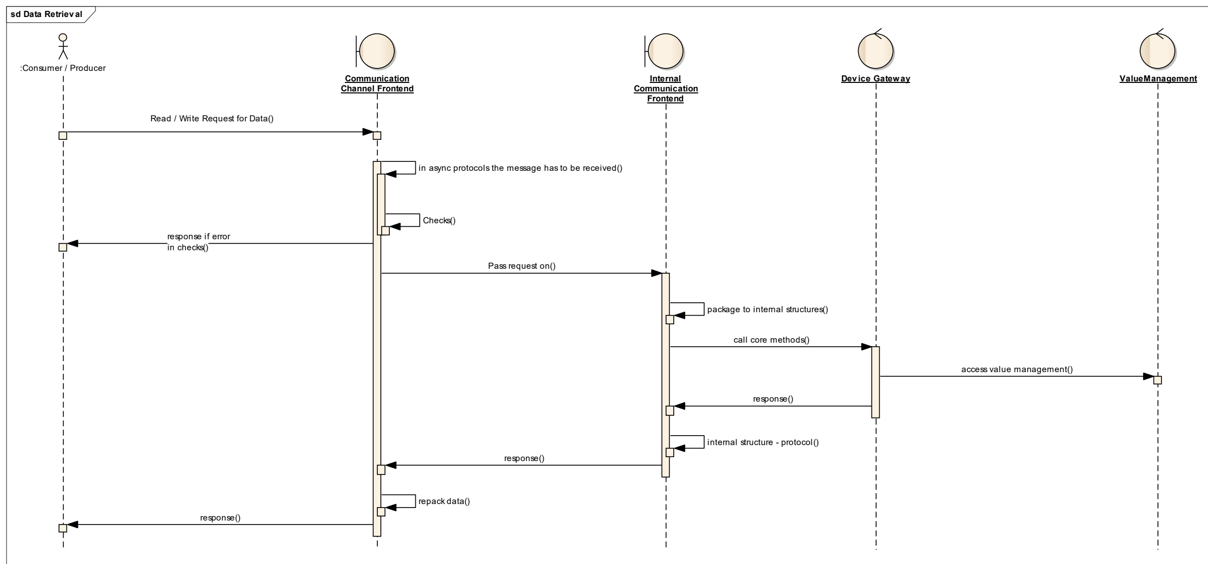

Figure 4.7: Retrieve values as a consumer

Usually it is an entirely synchronous process; only in case of asynchronous protocols (like for example message-based ones) this might imply an asynchronous element. For example, for SOAP or REST the call would start at the consumer and go all the way down to the *ValueManagement* with results flowing all the way back.

4.5.6 Data access

Sensor or global data of the device gateway has to be stored in permanent storage and accessed there to be usable. This access is designed to be handled by subclasses deriving from an abstract class *DataStorageAccessBase* which requires the derived class to implement methods like: *StoreDevice()*, *StoreSensorDate(), GetSensorDependencies()*, etc.

All methods needed by any module within the device gateway will have access to these methods and no other data access methods should be used throughout the system.

To use an abstract base class in the design allowed the provisioning of a static singleton[96] property as a concrete factory implementation (which an Interface cannot provide). In the singleton the *ConfigurationManager* is queried for a concrete implementation class which is then instantiated. Should there be no other

[96] A singleton is a design pattern which allows to make an instance of an object globally available and guarantee that only one instance of the class is actually instantiated (https://msdn.microsoft.com/en-us/library/ms998426.aspx)

implementation, the default one would be for Microsoft SQL Server. Figure 4.8 shows the UML class diagram for this mechanism.

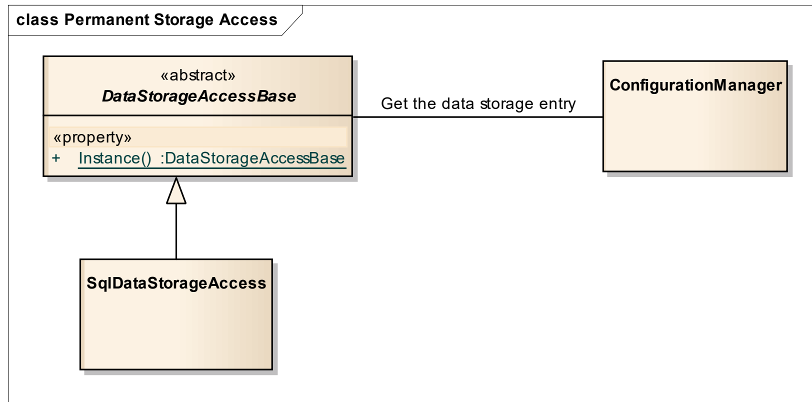

Figure 4.8: Data access class diagram

In this way it is very simple to exchange the permanent storage access to for example another SQL database like ORACLE, mySQL, etc. or even to a flat-file system or a hybrid approach (for example store sensor data in flat files or ISAM and the rest in an SQL like structure).

Changing the implementation should require an effort of perhaps 8 hours for an experienced developer and that way it can be considered easy to change.

The ability to use alternative storage systems is very important as many integration scenarios either have existing database infrastructures, or require versions, which are royalty free[97] or want to use the fastest possible access to the sensor data, which might be flat files or ISAM like systems. In addition, as NoSQL[98] databases become important for large scale data systems, this way integration into those systems is possible as well.

4.5.7 Value management

The value management of the device gateway is one of its core components and of paramount importance to the whole system as everything regarding sensor values depends upon it. Several areas are covered by the *ValueManager* component, which are described in subsequent sections in more detail:

- Handling of the current sensor values (read / write) and their history
- Populating the sensor history during start of the device gateway

[97] Which the standalone SQL Server version would be as well, as long as the data is not more than 10 GB https://www.microsoft.com/web/platform/database.aspx

[98] https://en.wikipedia.org/wiki/NoSQL

- Callback handling
- Dependency Management
- Virtual Value Evaluation including asynchronous (cyclic) modes

4.5.7.1 Handling of the current sensor values (read / write) and their history

As the *ValueManager* component has to provide quick access to the values (usually the current) of sensors it must organize them in a way so that they are quickly and efficiently accessible. This is especially important as this operation is often called synchronously to other operations, so a lengthy delay to for example load data from a data store is not usable. It is assumed that any implementation provides:

- Direct access to the current value of any sensor
- Direct access to a suitable (depending on the use case) sized set of historic values and then to a value within this set
- Access (with additional loading) to any historic value of any sensor
- Virtual sensor values

The current value is thereby defined either as:

- The last reported value either by scanning for a value or as it was sent from a sensor to the gateway
- In case a definition for a value expiration threshold exists (*DataValidityThresholdInMsec* in the sensor registration) then the current value is only defined, if the time difference between request and last update is less than the threshold
- If no current value exists, a configurable default or a NULL-representation (no value present) is returned (it is configurable what should happen)
- If the sensor is a virtual sensor the virtual value evaluation will be used to determine the current value, except if a value exists and the virtual sensor definition does not state "calculate on request" as mandatory. In these cases, the existing current value will be returned.

In case a historic value is accessed it is either served from the cache, or, in case not present, from the query result against the data store.

When writing the current value several call-backs are performed to allow for checking and potentially adjusting the value. In case this succeeds the current value is supposed to be over-written, the last update synchronized and a historic entry added.

In case the sensor is defined as "sensor data to be written to data store" this has to be done. If in addition the flag "direct persist after change" is given, the writing must happen synchronously to the change, which makes it a very expensive operation.

Should the sensor be an actuator, then the value will be propagated to the actuator by means of a background thread. In case of a synchronous operation the write should be enqueued at the head of the request queue to not block the value management. This enqueue at the head of a command queue between threads is a general design pattern taken in the value management as it is of paramount importance that operations perform as swiftly and least intrusively as possible, as access locks (semaphores) have to be kept on each sensor value to prevent multi-threading problems, which makes synchronization already difficult. In addition, the latency involved in "out-of-bounds" calls would make any predictable access difficult.

The specific actions and flows to handle the current sensor value are described as an UML Action diagram in Figure 4.9:

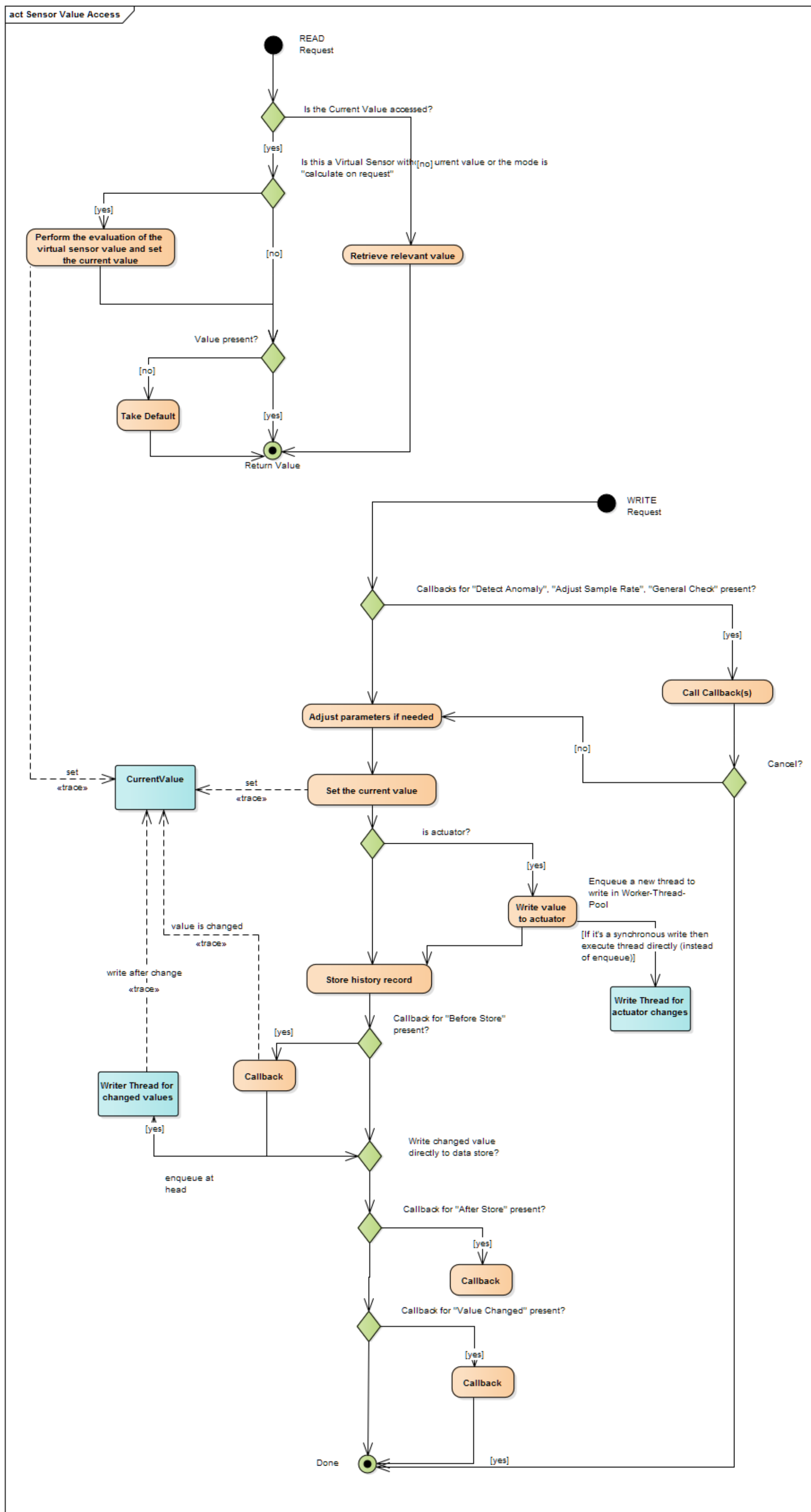

Figure 4.9: Current Sensor Value Management (read / write)

Specially to gather information about timing, limits, etc. the tests "Timing to push values into the core with dynamic calculations" (section 6.2) and "Concurrent access (READ / WRITE) by several clients to check for concurrency, race conditions and locking issues" (section 6.4) have been designed.

4.5.7.2 Populating the sensor history during start of the device gateway

To enhance system performance, it is recommended that during start the device gateway pre-loads a suitable set of last historic values for sensors from the persistent storage (database) into runtime memory. This is especially important if triggers (or consumers) access these values quite frequently, as otherwise a constant loading from the data store happens, which, due to latencies, can cause overall system degradation.

Assuming an approximate size of ca. 100 bytes for a value entry (data + associated control information) and 100 historic values / sensor this yields 10 KB for each sensor. Thus, even with 1,000 sensors the load factor would only be 10 MB which is no concern at all. Therefore, the design considers pre-loading all existing data - at least up to reasonable sizes.

4.5.7.3 Callback handling

Callbacks are used for several things inside the value management where the defined types and their use[99] is defined in Table 4.2. They are designed like function calls where the callback is called (according to the type in different scenarios), gets the current value and environment (to access historic data for a running-means for example) passed in and returns the new value. In addition to returning a value the callback can signal if the current operation shall be cancelled or the sample rates adjusted.

[99] Should the need for additional callbacks arise this is simply a matter of extending the defined callback types and call these types in the appropriate places.

Table 4.2: Callback type and use

Callback-Type	Meaning / Use
DetectAnomaly	The callback is intended to check the incoming data for improper values (anomalies). Usually the value would then be discarded
AdjustSampleRate	Check if the sample rate has to be adjusted by either giving a higher or lower rate
GeneralCheck	Check the incoming data in general. Does it fit the intended data type, is the value ok, does it match the series, etc.
BeforeStore	Before a value is stored this callback can be used to manipulate the type
AfterStore	After storing is done. This is more a "marker" callback which might write the value to for example a second system, etc.
VirtualValueCalculation	The callback is intended to calculate the virtual value for the current node
AfterChangeCallback	After a value change where the change amount is configurable (from “always” to a specific value)

Here DetectAnomaly, AdjustSampleRate or GeneralCheck Callback-Types are used to validate and / or modify data before it is accepted into the device gateway value management.

The AfterChangeCallback Callback-Type which will be called after the value has changed in the value management. In the handler any action could be done – including sending for example notifications to other systems, etc. The default implementation would be to provide feedback to a URL using a POST operation.

To use workflows as callbacks especially for value change is a powerful, yet very (potentially) dangerous operation as well. Workflows can be very unpredictable as their timing usually depends on many factors and especially long-running tasks pose a big challenge to the device gateway infrastructure (serialization of state, wake-up and continuation of run, etc.). Therefore, the device gateway by design does not support suspension of tasks (thus they have to start and finish in the same iteration) and has to run them in a dedicated secure space so that, in case of trouble (for example a

watchdog expiring), the whole operation can be stopped without damage to the remaining system.

All callbacks are stored system-wide in a component called the value manager which manages all values and all callbacks currently existent.

Further information about callback details can be found in section D.4.2 (appendix).

4.5.7.4 Virtual Value Evaluation including asynchronous (cyclic) modes

The evaluation (computation) of a virtual value is a straightforward process which depends entirely upon the existence of the callback to compute the value. It can be either executed synchronous to the request or be performed asynchronous automatically in a cyclic background operation[100].

When a virtual value is requested (synchronous mode) and a computation rule exists, as well as the value is designated as to be computed on request[101] and no threshold value is present, then the callback for this value is loaded and executed.

In case the computation was not cancelled by the callback the new value is returned, otherwise the existing (or default, in case none has been established beforehand).

These operations are shown in the UML activity diagram in Figure 4.10.

---

[100] This option is usually relevant if the basis for virtual values does not change too much in a given timeframe, or the virtual value does not have to 100% reflect the underlying value like for instance the average room temperature where a fluctuation of 0.1° C does not make a big difference

[101] Other options would include to be computed by a cyclic task or when the data item the value is dependent upon changes

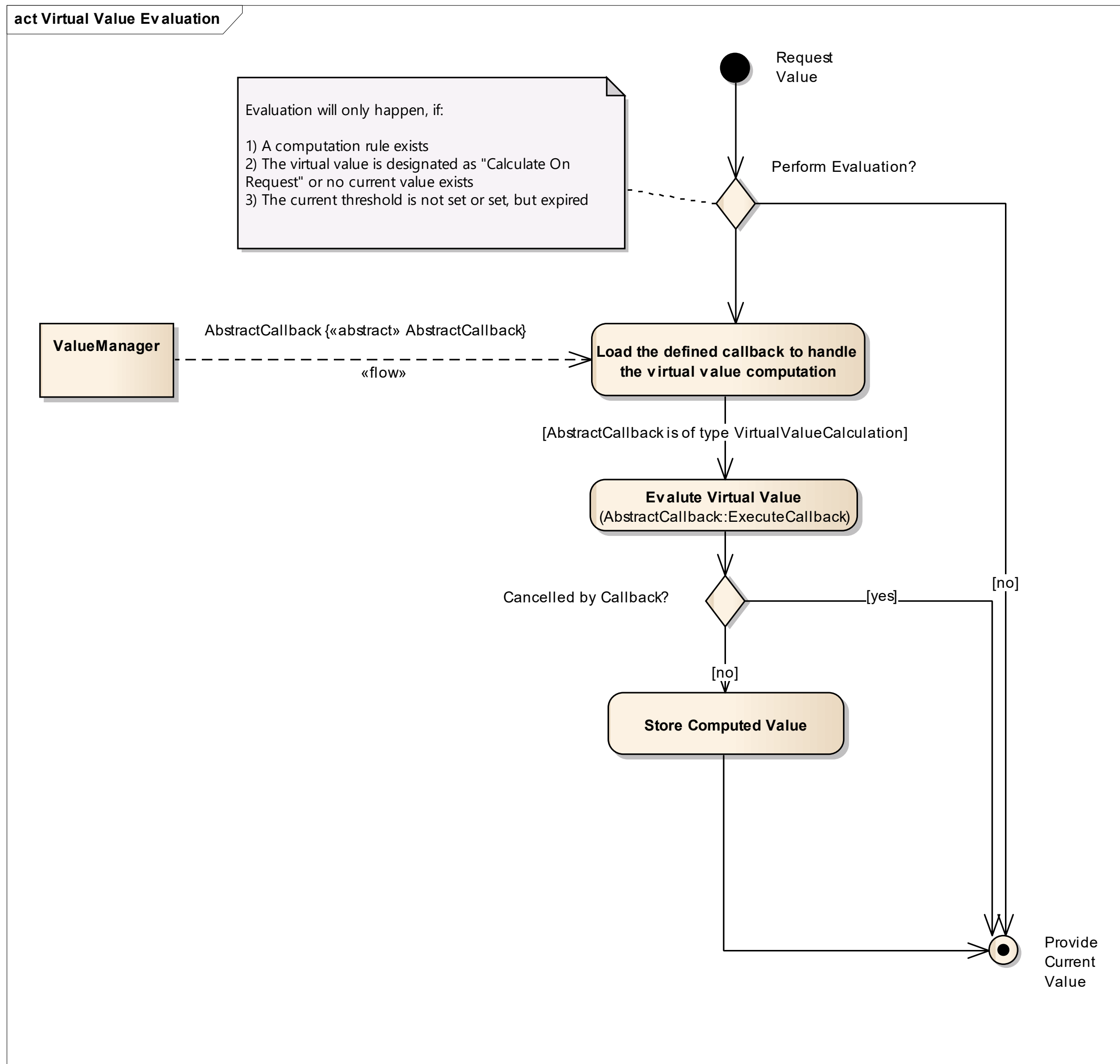

Figure 4.10: Virtual Value Evaluation

If a background (cyclic) evaluation is set up, then a background thread will be started, which checks every x millisecond all virtual values and re-computes them by evaluating their value[102].

4.5.7.5 Dependency Management

Another task which has to be performed by the value management is the dependency management between sensors, virtual sensors and their dependencies. The core tasks here are:

- Register dependencies and persist them in the data store (using data access methods)

[102] Future versions might improve this by providing several parallel background tasks with for instance different evaluation times

- Check for cyclic dependencies which are not permitted

The cyclic dependency check can be done by simply building a tree of all classes which constitute the dependent upon (base) type and then check this tree for any occurrence of the dependent type in it. In case a cyclic dependency is discovered this has to be rejected.

4.5.8 Watchdog and cyclic execution with workflows

To be able to perform additional house-holding tasks which are usually always present in an integration project, especially when doing business-process-integration, the device gateway is designed that a background task exists which is able to execute other tasks on a cyclic basis.

For this the task to be executed has to be generally registered in the device gateway registration and implement the interface *ICyclicTask* which is shown in the UML class diagram in Figure 4.11.

The *CyclicTaskRunner* component then scans the configuration for all relevant entries and for each entry loads the task (if not still ready from a previous run and having the attribute *KeepAliveBetweenInstances*). Then the task is executed and the next task used.

This can go on as long as the server is running.

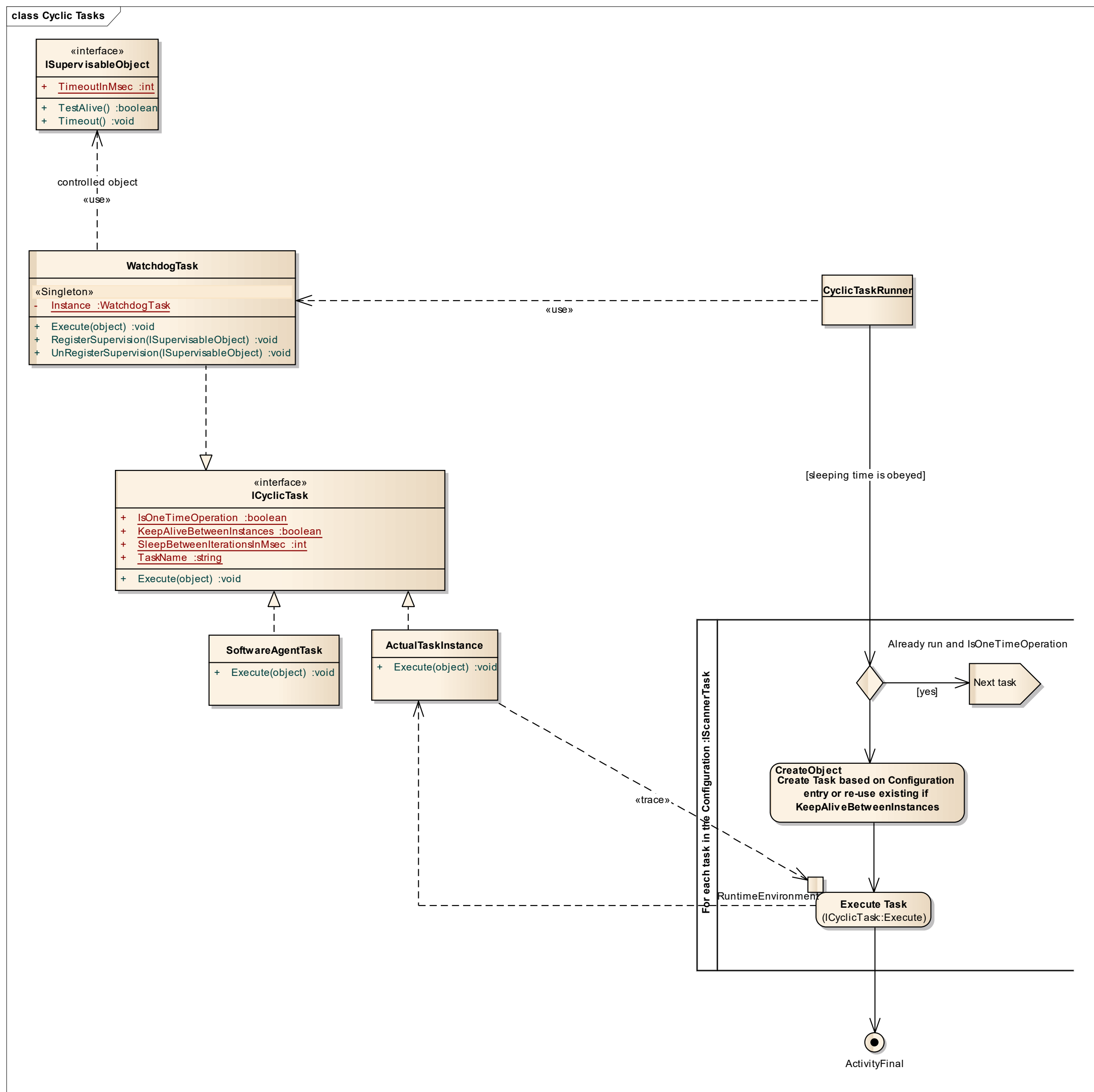

Figure 4.11: Cyclic Task Execution

4.5.8.1 *Watchdog task*

A special task always running is the Watchdog task, which can be used to supervise operations in various components. Any component or object implementing *ISupervisableObject* can register in the *WatchdogTask* and then the watchdog will check that after each object's n msec the object is still alive by reacting to the call to its *TestAlive()* method. In case an object is no longer responsive the *Timeout()* method has to be called which will have an implementation inside the object and there any further action can be taken (un-blocking communication channels, etc.)

4.5.8.2 *Workflow execution as a task*

Very similar to the execution of workflow callbacks in value management workflows can be used (and executed) as tasks as well. The difference here is that now long-

running tasks can be supported as well - including suspension and re-animation of a suspended task. Especially for these workflows is a provision in the value management so that correlation identifiers (ids) can be associated with values that on arrival for example of a special value, the correlation between value and workflow can be established and the agent continued.

This implies hosting of the workflow runtime environment and execution engine as well as providing additional support for suspension (hibernation) of agents and re-animation.

4.5.9 Access control

Access control is designed to act as an empty shell at the moment (at least for the reference implementation), which can be extended to a fully functional security integration with full access control if necessary. The UML activity diagram for this topic is presented in Figure 4.12.

Every method and communication entry point reachable from outside will branch to a centrally configurable loadable security context provider which is then responsible for letting the request pass or fail (usually be means of an *AccessNotAllowed* exception and a corresponding log entry).

The *DefaultSecurityProvider* does nothing and lets every request pass so that every consumer can access any data present in the system as well as insert any data for any device. This decision was made deliberately as for most integration environments an existing security infrastructure has to be used, which then requires adaptions and further integration.

If such an access control is needed in a specific business process integration environment then a specific *SecurityProvider* has to be implemented, which checks the object passed in, which is of interface *ICallContext*. Using this interface, the implementer can extend the call context as well (by having specific classes derive from it). This is important as depending on the security infrastructure different parameters will be needed and passed. Some store for example the user as an ID in each web request in a request variable, others have an LDAP id, etc. In addition, as the system cannot know which operation has to be tested, this will have to be passed in the current context as well (for example "CREATE-SENSOR", etc.).

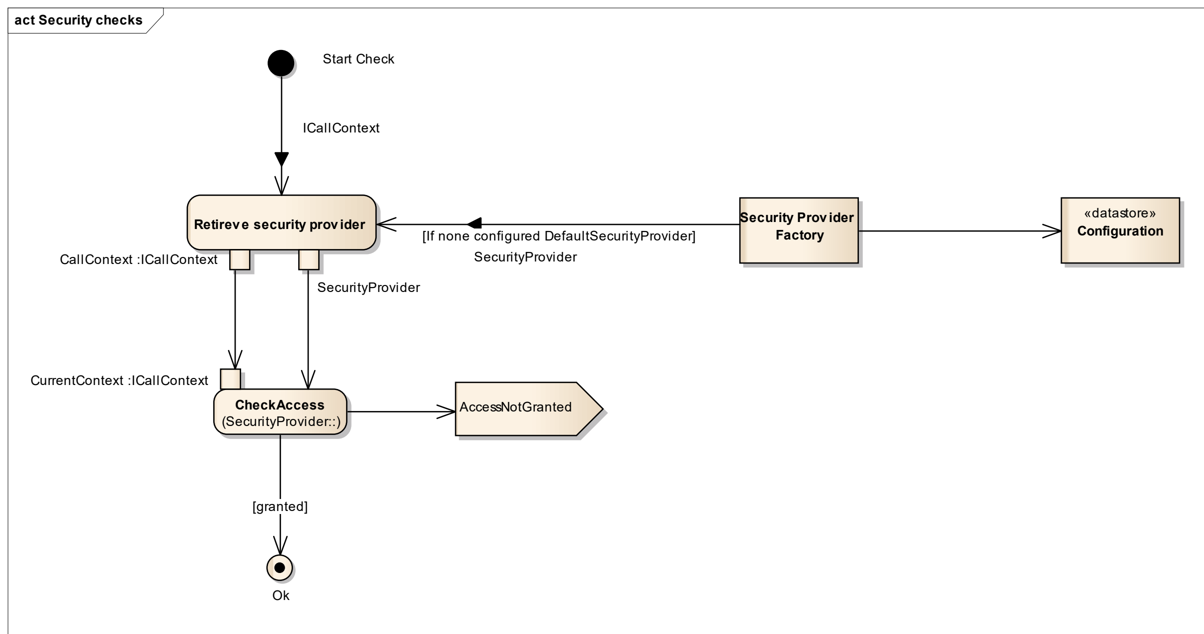

Figure 4.12: Access control

4.6 Physical view

From the physical point of the design, principles of the device gateway were:

- Being modular in every aspect from software development to physical deployment
- Allow the utilization of existing infrastructure which is proven and "already there" (do not re-invent the wheel)
- Be as small in terms of "footprint" and easy to use as possible. In a typical usage environment everything should run on one physical (moderately dimensioned) machine

Database, communication handling, inbound and outbound requests can all be distributed to different physical systems with internal binary communication between them. Only, due to the stateful nature of the "Value Management" (section 4.5.7) with all associated triggers, virtual values, etc., the device gateway core (with all associated components) has to be on the same (and single) system and running permanently as an independent process.

To design the DBGA as a classical request – response web-service seems not very useful as then state data (current values, dependencies, etc.) would have to be retrieved from the database for each request. To just retrieve a serialized state (for example from a file on disk) is not possible as it might happen that several instances of a web-service (perhaps even on different nodes in a cluster) are instantiated at the same time and then all serialized states would be invalid as none reflects the overall system.

Therefore, a re-build of the state is necessary for each request. This is not very efficient and uses vast system resources for each request whereas in a stateful system data is preserved between requests and always available. In addition, the internal communication protocol would then have to be either REST or SOAP, which implies significant costs in terms of processing time and memory consumption versus a binary protocol.

In addition, dynamic processes like value change handling would be much more complicated as issues like several value writes for the same endpoint which are routed to different instances by a load balancer have to be resolved.

In case of need the device gateway could be split across several physical machines, but it must be guaranteed that sensor values dependent upon each other are kept on the same machine so that updates and evaluations work.

Figure 4.13 shows a typical simplified physical view (with the assumption to have a Windows execution environment[103]) of a device gateway system with all major components included.

[103] Should another system be used, then IIS must be replaced by another hosting environment

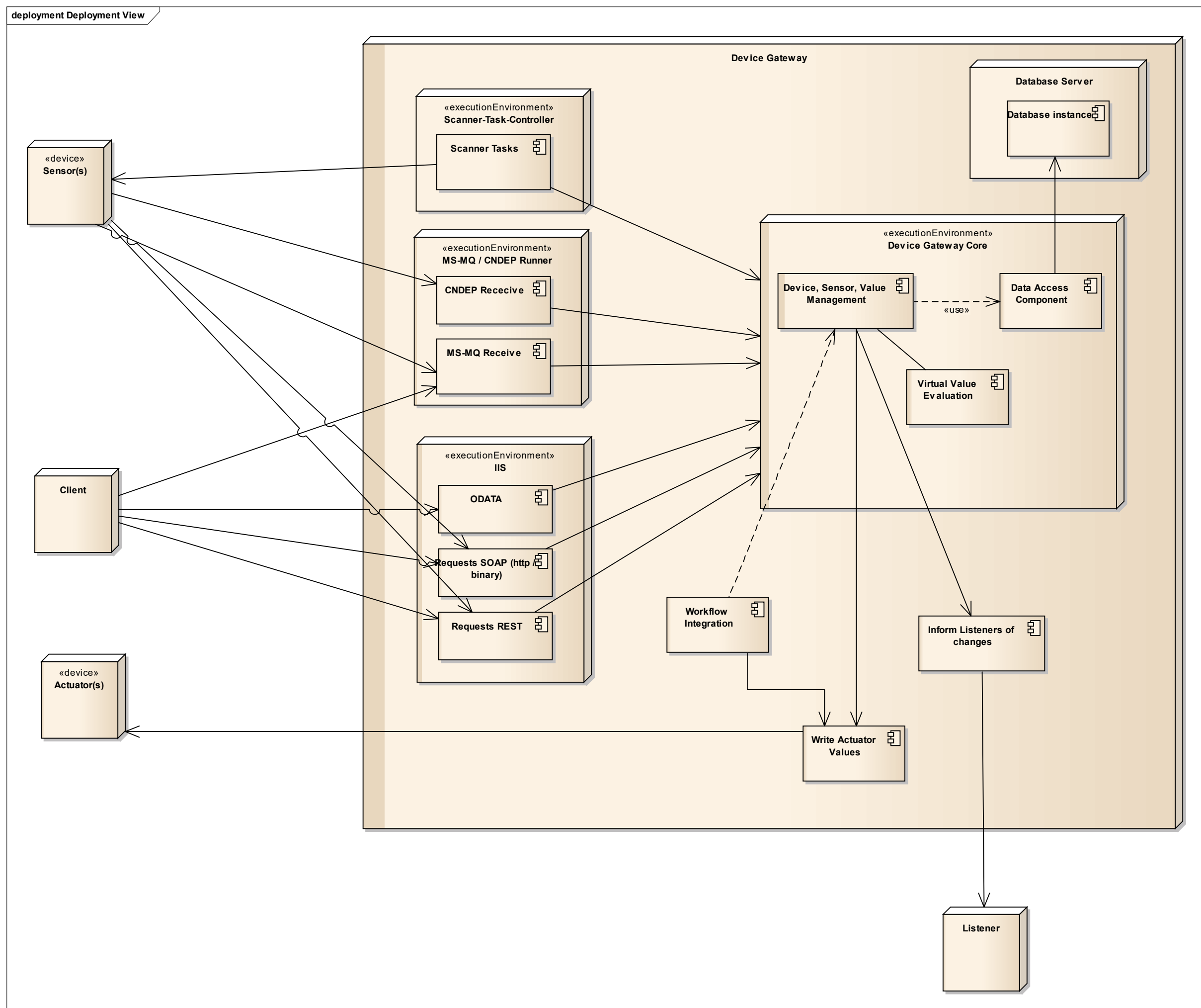

Figure 4.13: Exemplary physical view of device gateway

Whether the hosting container for ODATA, SOAP and REST is IIS or any other available option does not matter and only might have impact on the processing time requirements and memory consumption.

4.7 Data model

The data model, where details about the physical and logical design can be found in the appendix (section 0), of the device gateway is very simplistic and designed to be extended[104] when needs arise, yet act as a fully functional model from the very beginning.

It focusses around the following general data structures:

---

[104] by means of a pluggable architecture where the provider of the data storage can be configured and provides common methods. See section 4.5.6 for details

- Device and sensor / actuator and their relations to describe which sensor / actuator belongs to which device and has which relation to other sensors (forming chains) including some physical attributes (like IP-address, etc.)
- Some additional semantic information about the sensors and actuators to provide a minimal level of ontology support
- Sensor data (either from a sensor or calculated)

Therefore, much additional information usually associated with sensors or devices is omitted as this additional data is usually only needed for discovery of devices (based on additional data attributes) and reporting.

Especially classic markup-languages like SensorML [71] or other XML based variants [58] tend to be all-encompassing, yet on the downside huge and hard to implement. For most business-integration scenarios a much simpler approach is sufficient, yet when it comes to for example world-wide distribution and querying of sensor-data, such an alternative approach is more suitable. Due to the pluggable architecture an implementation of such an approach is no problem, as the core definitions of what a device and sensor is can be exchanged, as long as the functional elements like virtual value evaluation, and so on are left intact.

The current design has no precautions for data aggregation after some time x, etc. This is usually done on a project specific level (or as part of a big data solution) and can be handled perfectly in an integration as a stored procedure; in any case not as a core task of the device gateway.

4.8 Discussion

The data source and destination management and registration process design (section 4.1), the gateway communication design (section 4.2), as well as the use of workflows in device gateways (section 4.3) was described. This was followed by the matching of use cases to involved components as logical view (section 4.4) and the process (or dynamic) view in section 4.5, followed by the physical view (section 4.6) and data model (section 4.7), as well as a final conclusion in section 4.8.

Having defined the design elements that make up the complete design, next is to describe the reference implementation of the design. This is presented in detail in chapter 5, which uses and applies these design elements.

5 Reference implementation of DBGA

This chapter describes the reference implementation, by discussing the implementation approach chosen (section 5.1), the general structure of the implementation (section 5.2), sections 5.3 to 5.8 are then devoted to describing individual components, and finally section 5.9 provides reflection on the experience of creating the reference implementation.

The reference implementation that has been implemented is a fully functional implementation of the design of the proposed Device-Business-Gateway Architecture (DBGA) described in chapter 3.7.

The reference implementation (which is available to download[105]) was used:

a) As a test-bed to perform the research experiments (see chapter 6) for checking that the design is stable, sustainable and that it fulfills the requirements/characteristics outlined in chapter 3;
b) To implement the case studies (see chapter 7) as a proof-of-concept and further check of the achievement of the requirements/characteristics;
c) To provide a solid basis for anybody wishing to use such an architecture for their own device integrations.

The reference implementation was undertaken using the following technology stack:

- .NET 4.6.1 on Windows 7 Enterprise Ed. (and Windows 10)
- Windows Workflow Foundation 4.5
- SQL Server 2012 Express Edition[106]
- IIS 7
- IronPython.NET 2.7

Each of these components, aside Microsoft Windows which is a commercial product (and contains IIS 7), is available as either open-source or free of charge.

In the opinion of the author of this thesis, the choice of these environments and components guarantees a thorough and stable platform, whilst also exhibiting a very modern programming environment.

---

[105] https://github.com/mglienecke/DeviceGateway

[106] Whereas the Standard or Enterprise Edition work 100% similar

5.1 Options of how to implement the reference implementation

Based on the characteristics defined in section 3.5 and the components described in section 4.5 there were a number of choices as to how to implement the reference implementation:

1) Create a monolithic application which includes all logic and handling in one large executable piece of software;
2) Create a pure web-service-based software solution;
3) Create a hybrid solution with a computational core and exchangeable interfaces / services.

Approach 1 usually tends at least in the beginning (based on the experience of the author) to be the quickest and fastest implementation approach, yet over time this changes as the effort expended to maintain such applications usually increase quite a lot (compared to more layered and disperse applications). In such an approach you normally find the lowest degree of complexity as very little interaction between components outside the single process takes place and in general it is straightforward to develop. Yet for the anticipated environment and requirements, a monolithic application might be usable as a reference implementation, but would not be usable anywhere in a real-world environment for the following reasons: it would not scale, would be hard to expand and in general very would be inflexible. In addition, it would be a contradiction to modern software engineering, especially modular and extendable design.

Such a design approach allows for exchange of parts and components and replacement or extension of them with better, corrected, enhanced or somehow different versions, without re-distributing the entire application. Redistribution of applications can be a major issue and therefore a modular approach would be preferred.

With Approach 2 every part is modular, exchangeable and all things work together by communication links (URLs), which can be configured. If a service A needs some feature X it can obtain this from another service which publishes the interface and can consume and handle the corresponding requests.

There is just one major obstacle to this – maintaining state. The maintenance of state requires some active part, which monitors the state and reacts to changes. As web-services are always only passive (they are called and react solely on this activation – afterwards they are suspended, kept sleeping, etc.) they cannot perform that task directly but always need such an active component. It might be an option to use for

example a cache service or a SQL database to maintain state, but this would require quite a number of transactions and interactions and thus could make the system quite complex and perhaps slow as well. This passiveness is an even bigger problem when it comes to autonomous operations when some action (for example checks) must be performed without any external trigger or stimulus to start an operation.

Even when using workarounds to alleviate such issues a purely web-service based solution usually is quite complex as many components are interacting to create the desired result.

To compensate for this, Approach 3 was chosen as the basis for the reference implementation. Such a solution approach has proven successful in the experience of the author in terms of stability, manageability and especially effectiveness of implementation as it combines the swiftness and ease of approach 1 with the extendibility of approach 2. The implementation uses a centralized process running as a core (this can be run as a system service as well) which utilizes several communication services to handle client requests (consumers and devices / sensors).

It should be clearly pointed out that the same solution of course could be built using a pure web-service approach with some added active components (acting as triggers to start web-services, etc.).

In a use case where distribution, clustering or fail-over-resilience of the system is important, the following considerations should be considered:

- The database server can be operated completely as a cluster with full resilience
- The communication frontend: ODATA, SOAP, REST handlers, CNDEP, MS-MQand scanner tasks can each be put on individual machines as the communication internally can be done over a network as well
    - The IIS part (which in typical environments will take the most load) can be (due to its stateless nature) completely operated as a cluster with full resilience

5.2 General structure of the reference implementation

One of the core considerations, when implementing the components presented in chapter 3.7, was how to efficiently package and combine them in terms of modules, assemblies[107] and general structure.

A big problem was that some internal classes – mainly the data structures needed for communication in SOAP and objects used in JSON based communication - are needed in the server- as well as the client-side. In addition, if the client is a .NET MicroFramework client[108], the format for the assembly is different and thus the code for these platforms has to be generated from the same source, into a different target environment.

One approach could have been taken would be to have many small assemblies, each resembling a single component, another to package the components into logical groups and combine several of them together.

Therefore, based on author's experience from similar projects as well as practical field tests[109], the components were packaged into the following logical implementation components, which are described in more detail in subsections 5.3 to 5.8 of this chapter:

- *CentralServiceLauncher* which launches the server-side process and starts initialization;
- *CentralServerService* acts as the core of the whole device gateway. Here all internal logic is implemented; Internal to this component are several sub-components (for details see section 5.4): ValueManager, Communication Modules for CNDEP and MS-MQ, Actuator Writing and watchdog as well as cyclic task execution;
- *GlobalDataContracts* provides all system-wide definitions and declarations for data-types, enumerations, and so on;
- *DeviceServer.Base* is a reference implementation for a basic server process running on a .NET MicroFramework enabled device;

---

[107] An executable module in .NET - typically in form of a DLL

[108] This was a big consideration for the *DeviceServer.Base* implementation (see later) on the .NET MicroFramework devices

[109] Especially the Micro Framework due to its different internal format of assemblies and serialization required a lot of low-level try and error work

- *DeviceSimulator* is a fully working device server as a simulation process on the host system for easier debugging;
- *GatewayServiceContract* as IIS-plugin modules to allow REST- and SOAP-handling;
- *ODATA* is a handler to process ODATA requests.

These components and their interactions are shown in Figure 5.1 and described in detail in subsequent sub-sections.

The main goal was to have as few as possible, yet as many reusable components as possible without having one big "mega-assembly" and "1001 tiny assemblies" as extremes.

In addition to the previously mentioned main components, there are several other components provided:

- *ODataConsoleConsumer* as a process which requests and consumes ODATA from the device gateway and therefore can be used as a debugging tool (versus using Excel);
- *PerformanceCounterSensorTask* as a sensor data producer task which scans Windows Performance Counters and passes them as sensor readings into the device gateway. This acts as a sample of how to actually scan for values and submit them, in addition it provides added benefit as arbitrary performance counters (by means of configuration) can be passed to the gateway and used in for example decisions;
- *MsmqSensorTask* which transmits data for arbitrary (configurable) sensors using MS-MQ and accepts values written back as actuator values. This can be used as a test-bed to see actuator values being passed back (for example from a workflow);
- *UnitTests* to excessively test all components and behaviors (as regression tests as well).

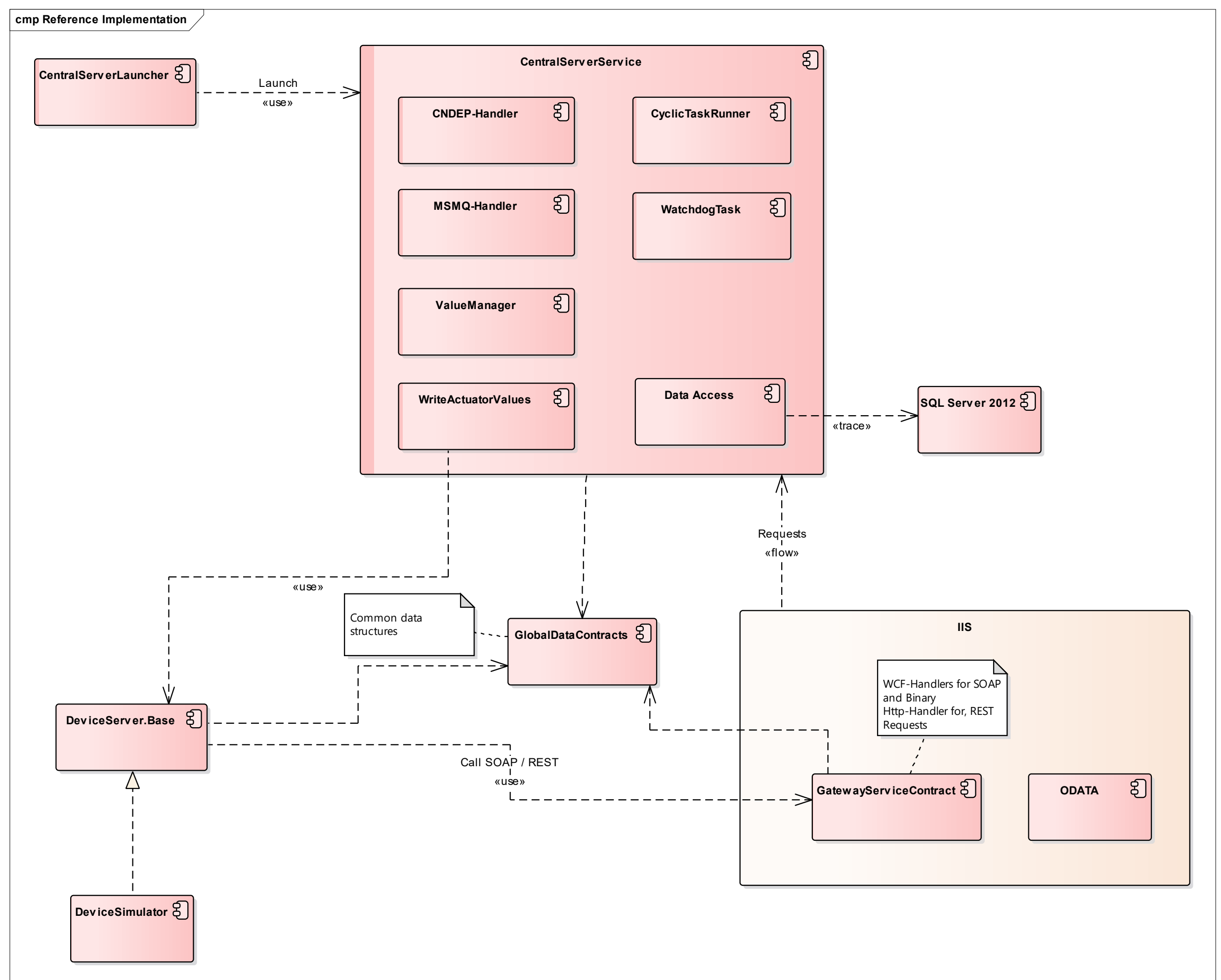

Figure 5.1: Reference Implementation (Main) Components

In the following subsections additional information about the implemented components and their relation to the design can be found.

5.3 CentralServiceLauncher

The *CentralServiceLauncher* is the launcher to start a device gateway server, which must be running at least on one machine in the network.

It acts as the runtime and launching environment for the *CentralServerService* (section 5.4), which, being an assembly, cannot be started directly. When started, the process contains the loaded *CentralServerService* and the device gateway server is fully operational.

In addition, to provide better support for debugging and testing the launcher can be started as a foreground command line process and all output is directly visible on the console, whereas in normal operation mode, it would be started as a Windows service and running in the background (without visual output).

Running as a foreground process has the advantages that all output written to the Console (including debug output and other error information) is directly visible so that a setup can be easily controlled. Later on, when the implementation is stable a switch to a background operation (as a service) can be easily achieved. The background service has the advantage that it can be controlled by the Windows Service Manager, started automatically, declare dependencies[110], and so on. The dependency declaration is especially very helpful as this ensures that for example SQL Server (which is necessary for the implementation to start-up) is already running.

5.4 CentralServerService

At the heart of the entire reference implementation is the *CentralServerService* component which is implemented as an assembly. This assembly will be launched by the *CentralServerLauncher* process either as a normal foreground process or as a Windows Service (invisible in the background). It implements several components which are detailed in the following sub-sections.

5.4.1 Modules and components

Inside the central service various modules (either implemented as independent, reusable assembly or in the *CentralServerService* assembly) are contained which:

- Manage values and callbacks including writing the data back to disk;
- Write values back to actuators;
- Perform the execution of cyclic tasks;
- Allow the management of Watchdog timers;
- Run workflows as Windows Workflow Foundation Workflows;
- Handle requests from CNDEP and MSMQ (the Microsoft implementation of a messaging system);
- Provide storage access to SQL Server 2012.

5.4.1.1 Data Access

Storage access to SQL Server 2012 is provided in the assembly *DataStorageAccess* where 2 concurrent implementations are available. The one to be chosen for the actual instance can be configured in the app.config[111] of the device gateway.

[110] Done in the registration as a Windows service

[111] The standard way to configure application settings in .NET

One implementation is implemented using "classic" ADO.NET[112] which is very fast (among the fastest possible access methods), the other is done using LINQ[113] to Entities.

The first approach is far more procedural and contains many manual data copy operations (which are error-prone) than the LINQ to Entities which is much higher-leveled[114]. Yet as the overall speed benefit[115] is much higher for approach one than the better readability it was used in the reference implementation.

Storage access is very important for the overall performance of the system, but as most values are cached in any case in the value manager there is not too much impact. The only exception being the part when value changes have to be written directly to the data store - here a method *StoreSensorData()* is called which always assumes that several data items are stored at once and therefore puts them in a temporary table with a bulk copy operation afterwards. The bulk copy prevents some internal features of SQL Server but makes the execution much faster. As usually the writing of sensor data is undertaken asynchronously in the background and therefore several hundreds or thousands of values have to be written at one, this is much faster than opening a transaction, looping over all values, performing the write operation and afterwards undertaking commit (or rollback). Tests have shown that the bulk copy performance vs. transactional one-by-one is approx. 100 times faster which makes it worthwhile to use.

5.4.1.2 ValueManager

The *ValueManager* component is the complete implementation of the *ValueManagement* component (section 4.5.7) and contains everything in this area from handling the initial loading, managing the sensor data, perform virtual value calculation, validation and callbacks as well as notifications and workflow integration.

---

[112] Microsoft technology to access data sources. It is a bit old-fashioned but still widely used

[113] LINQ = Language Integrated Query; an extension to C# where SQL-like query syntax can be embedded into the C# code which is transposed by the compiler to the calling of extension methods on the base objects

[114] And much more object-oriented – a real ORM (object relational mapper)

[115] The direct access is roughly 2-3 (in extremes 100 times) faster than LINQ. Especially larger INSERT-sequences cause a big problem here, which could be alleviated by using 3rd party libraries which allows “BULK” INSERT operations into the database (with all the problems associated), similar to the approach chosen

It is implemented in an independent assembly using a singleton pattern. This availability as independent assembly was important as now independent load and unit tests could be directly performed against the component without having another context around it. In addition, due to this implementation approach, the value manager is 100% re-usable in other implementation environments, so in case an integrator would like to change the central service, the value management could still be used as is.

As the *ValueManager*, with 35 – 45% of the total CPU cycle consumption in typical scenarios[116] is the single-most consumer of cycles any optimization and performance check is beneficial for the complete device gateway.

Inside the *ValueManager* all defined sensors (for all devices) are kept as a dictionary of value definitions with the sensor-id as a key. Inside such a value definition is:

- The definition of the sensor (virtual value, formula, etc.)
- Dependencies (if present)
- Current value
- Historic values

The historic values are set from the loading module (the *CentralServerService*) by calling a method during its own startup process. It has proven[117] to be better that the caller starts this population process which could be quite lengthy (instead of the component populating itself) as then overall system control is better coordinated. As only one method has to be called a change from the encapsulating component to the actual component would be very simple. In additional scenarios (when handling larger data than in the test environment which used 100,000 sensors with 50 values as in memory cache for each sensor as historic data as a maximum[118]) this task could be delegated to a background thread and then the load could occur in parallel to the remaining load process[119].

---

[116] Tests were done with a CPU execution profiler

[117] Separate test cases were made to check either approach with a CPU profiler

[118] According to the author's experience these sensor counts would happen very rarely in non-cloud-based solutions; 100 – 400 sensors is a more normal value. 50 values / sensor as direct in-memory history usually is enough as well, as failures or abnormalities should show up in such a sampling frame, too (of course depending on the sample rate, too).

[119] As usually only a small subset of all defined sensors is used this could be staged as well

Using a dictionary for the sensor access allows a very fast (by using the sensor-id as key there is simply a lookup operation with O(1) constant access speed), thread-safe access to the sensor definition and data. Especially the thread-safe access is of paramount importance as very often several requests for the same value from different clients produce overlapping requests, which are handled as separate threads in the communication layer and thus could create race conditions and / or invalid values as they are not synchronized.

To provide the highest possible flexibility for virtual value evaluation, the following (extendable[120]) options are implemented in the *ValueManager*:

- *SQL stored procedure*
  Here the evaluation is done inside the database as a stored procedure. This approach enables much faster access to historical data (for example for analysis), yet other computational tasks or access to outside resource is very restricted for overall database integrity reasons.
  In general, this evaluation is suitable for virtual values which rely heavily on historical / structured data. Execution is distributed by nature as the stored procedure is executed in the context of the SQL engine, which very often might be an independent system.

- *Python*
  In this evaluation method an external Python module is handed to the Python execution engine (provided as external module[121]) by the device gateway and all computation is done within it. Using Python as a general purpose programming language has the benefit of easy adaptability, it is easy to learn and the existence of a wide know-how pool.
  The down-side is that the execution is performed within the context of the device gateway and thus has the potential of a severe degradation of system performance in case too many evaluations are done.
  This evaluation method is suitable for more computing-intense and data manipulation oriented tasks like for example online analysis, running means,

[120] By means of factory patterns and late loading of components

[121] http://ironpython.net/

etc. of items whereas access to historic data is not of paramount importance[122].

- *Code from an externally loadable code module*
  This approach uses late binding to load an external module, which implements the necessary interfaces, into the engine and then executes the code. This approach is the fastest (in terms of execution time) and most flexible of all approaches (as every necessary logic can be implemented). Due to the software development lifecycle for such external code modules, it is the most involved evaluation process as well compared to the simple and easy execution of a scripting language like Python or JavaScript

Inside the value manager are two other important objects implemented:

1) The writing thread which writes outstanding changed data values back to the data store. According to the design the thread communicates with the rest of the world by using a FIFO-queue, yet due to the writing in chunks (as bulk copies) a dictionary where a whole range of objects to the be written can be taken easily has proven as a better option. In a dictionary of unsaved changes, no order can be preserved which would mean that priority writes do not get priority treatment, yet as all writes are done in one go this does not actually matter. The thread is designed in a way that a write usually only happens every x seconds, except when priority writes have to occur as the test then is changed. Currently the minimum frequency where a write can happen is 500 msec which can be changed by adjusting a constant.

2) The virtual value calculation thread which calculates all virtual values which are defined as "calculated by virtual value thread". The thread can be started / stopped from outside the *ValueManager* class by calling a method. The task is developed in a way that every x msec (currently 250[123]) a check is made if there

[122] Especially as every change back and forth from Python to the gateway engine involves quite a lot of overhead

[123] Usually 1000 msec is a good value for a sampling frequency in many not time critical processes according to the author's experience. If the check is made every ¼ of the that the probability is very high that calculations are done timely so the next request is served with proper values

are any pending calculations. This check evaluates all relevant virtual values and checks if any underlying value did change and in case yes a re-calculation is performed. Currently this re-calculation is an iterative process which could actually take quite a long time to finish so future enhancements would be: a) to move the calculation to a real external thread as it cannot be assumed that the caller called the method in a thread context b) to use the parallel task library of .NET which allows the distribution of the calculation process on all available CPUs of the target system. In case of parallelizing the computation, care must be taken that values which are dependent on other values are computed in a proper order - the base values first, then the dependents.

5.4.1.3 Write values back to actuators

The central server service provides a thread which currently registers a callback routine for each actuator inside the value management. As soon as the value changes the value manager calls back the callback and then a thread is enqueued to write the actual value, given that the last operation had no communication problem as otherwise another write usually makes no sense.

This is a slight difference to the design where a FIFO queue is designed to enqueue high-priority writes first and then low-priorities afterwards. According to some test results the approach taken is quite suitable for smaller scale implementations and much easier to manage as no FIFO has to be managed, no locks taken, and so on. The thread is enqueued and simply starts.

In case more actuators are to be served, a better scalable approach using a FIFO queue should be implemented (according to the design).

5.4.1.4 Perform execution of cyclic tasks (CyclicTaskRunner)

A cyclic task object (the *CyclicTaskRunner*) is provided which during startup loads all tasks to execute from the configuration and then afterwards each task object is instantiated.

In an endless loop with an appropriate[124] sleep of currently 1000 msec (configurable) at the start, the tasks are checked if they need execution (according to the design) and if yes, they are started accordingly by calling their *Execute()* method and passing in the

---

[124] As is a good default sample rate 1000 msec for not time-critical processes (according to the author's experience) the same is true for cyclic tasks. Usually there is no need that these sample / run faster than the sensor data is retrieved.

current runtime environment, so that the task can access all sensor values, methods, etc.

5.4.1.5 Management of watchdog items (WatchdogTask)

The *WatchdogTask* object is implemented as described in the design and launched by the central server service. It provides all components and objects with the methods to register and de-register for watchdog checks as well as calling the appropriate methods inside the registered objects.

5.4.1.6 MSMQ and CNDEP handler

The *CentralServerService* provides default implementations in dedicated threads for handling inbound CNDEP and MSMQ messages.

Normally these message handling facilities would be hosted outside of the *CentralServerService* (like the SOAP or REST handlers), yet as there is no framework to run them directly exists[125], the decision was made to include them in the core service.

The communication internally is done using .NET Remoting as well, with the exception that the IP access is performed using a loopback adapter in memory. Therefore, both handlers act very much the same as if they would be really "outside" of the core service.

5.4.2 Communication with the outside world

This *CentralServerService* is – at least in the end - accessed by any user using the .NET Remoting protocol, which is a very fast and efficient (typically binary) distributed object message broker implementation. Even if some consumer uses SOAP or REST or any other protocol, internally all messages are broken down (inside the specific protocol handlers) to the .NET Remoting protocol.

All designated *CentralServerService* object properties and methods are exposed as remotely accessible and any request made on behalf one of these exposed end-points results in the corresponding message being transported from the client to the core as a remote procedure call. As this implementation causes very little strain upon the runtime environment as well as requires nearly no development effort, it was chosen as a matter of choice for the internal communication. Another benefit of using this technology is that is has been used in many customer projects of the author with

---

[125] SOAP and REST are handled inside IIS. So in case MS-MQ and / or CNDEP should be hosted externally a plug-in for IIS would be required or a dedicated hosting process.

trillions of transactions and an extremely optimized memory, speed and runtime environment footprint.

The only real alternative would have been to implement the server as a set of web-services but this has a lot of problems associated as described in the design chapter. The main reason being the stateful nature of the value management which requires values and history to be present for efficient virtual value evaluation and callbacks.

There is a limiting factor in using .NET Remoting when it comes to fail-over systems and clusters as there is no failover routing possible. Therefore, it is impossible to have two (or more) systems to share load and increase availability out of the box. Custom solutions to achieve this behavior can be developed (and have been done), mainly using hot-standby technology where one server takes over the others operations as soon as the first one fails, yet all these measures are not very easy and optimal to use. So it has to be noted that using this implementation approach the user is limited to use the device gateway on a single machine. As the .NET Remoting is capable of spreading out communication threads on several physical CPUs this can be alleviated (at least in terms of performance) as there usually is enough resources available to handle these requests[126]. It is not unusual to see 100 threads in parallel actively handling client requests at the same time, thus optimizing delays in responses from databases, external resources, and so on.

As Microsoft no longer encourages people to use .NET Remoting (yet it will perhaps never go away) in the long run alternatives will have to be investigated. Investigating options in this thesis research showed again that the move to a stateless WCF-based service-model will require excessive serialization / de-serialization of the entire value management and centralized synchronization of the stored data among several instances. In total this might be a more flexible approach when it comes to large-scale scaling and fault-tolerance, but yet the added cost in terms of infrastructure, deployment and especially administration outweighs the benefits substantially (at least considering the environment where such device gateways usually are used).

When there really is a need to handle more than 1,000,000 sensors with 1,000 values per sensor (which is easily doable on a modern PC), then in any case a total different

[126] Which has been shown by the tests as well.

environment exists, where different patterns have to be applied and a move to an architecture like Azure IoT might be appropriate.

These numbers have been derived through experience in real projects by the author. Usually in an industry automation project there are 100 to 400 devices (at the most) with 1 to 3 or 4 sensors / device. To have 1,000 values of history per sensor would normally mean a history of 100 minutes (as a sample is very often done only once every 10 seconds) which is normally enough for an online statistical investigation by validation routines or workflows. If there is real demand for historic analysis than this usually is either kept outside of the gateway as separate task (directly operating on the database data), or the history is increased to for example 10,000 or 100,000 records. This is mostly a question of loading time while starting the gateway as the data simply has to be read from the database. The internal structures are capable of holding as much data as needed.

In case an environment consists of 100,000 sensors or more this for sure is a geographically wide-spread environment, which requires a totally different approach to data sending and analysis. In addition, writing will not be required most likely as these environments tend to be mostly read / submit data only and perfect examples for the use of message based middleware like Xively or Azure IoT[127].

5.5 GlobalDataContracts

In this assembly all data structures used for communication between data providers / consumers and the gateway are defined.

In case of SOAP, CNDEP and MSMQ these data structures are serialized to transport them over the communication link and de-serialized on the receiver side to re-constitute the objects involved. For serialization the standard .NET serializer for string results is used[128]. In de-serialization the string form is transported back to the internal binary object format.

Should REST be used then the content is transferred using JSON and serializing / de-serializing is done to / from a JSON string.

---

[127] Which does not imply that it could not be solved using the Device-Business-Gateway Architecture (DBGA). Just the provided logistics from the mentioned architectures is much more suited to such an approach.

[128] There are others for XML, binary, etc.

5.6 DeviceServer.Base

The *DeviceServer.Base* is an application which can run on any device supported by the .NET Micro Framework. It acts as a kind of generic framework where custom device applications can be built on top.

The *DeviceServer.Base* application is fully working including communication between the device and the gateway using HTTP REST and CNDEP as protocols[129]. Support is added to act as data provider (sending sensor data to the gateway) as well as actuator which can be sent to/from the gateway.

So in case in a real project someone needs a real device the *DeviceServer.Base* could be taken and in designated places some logic added to have a fully functional piece of device software.

In addition, there are specific implementations for Netduino boards and some other vendors to provide support for their dedicated hardware.

5.7 DeviceSimulator

The device simulator acts like a real device with REST-based communication. It can be queried to retrieve new values (PULL), sends values on its own (if started accordingly - PUSH) and acts as an actuator destination.

Therefore all typical operations associated with a device can be tested and easily simulated which is much more efficient than to load a device image to a device (even using fast USB) all the time.

5.8 GatewayServiceContract

This assembly acts like a façade for the internal .NET Remoting protocol towards SOAP and REST requests.

For SOAP which in WCF[130] is easily configurable to accept binary- or http-based (or both) requests, this means simply the implementation of the service contract where the interface for the implementation is annotated as *[ServiceContract]* and each method within with *[OperationContract]* with the corresponding message parameters to mark an externally reachable message based API. Special care was taken to compose the message parameters for the methods only with types which can be represented in

---

[129] Support for MSMQ is not existent and SOAP is only partially implemented in .NET MF

[130] Windows Communication Foundation

other languages (like Java or PHP) as well. Usually this exchange is an area of great distress in mixed environments as different tooling and development systems have different assumptions about how the WS* call should be performed and how the messaging is done. With the approach taken Java-tools can generate client stubs as well, so that Java client can use the device gateway directly.

The actual integration into IIS as hosting environment for SOAP is done by registering the service inside the IIS. Should IIS not be relevant a hosting inside a standalone SOAP-Request-Container would be possible, yet performance might suffer as these are not as optimized as IIS would be.For REST-handling the *GatewayServiceContract* assembly provides handlers for http-requests which are analyzed according to the verb (GET / PUT / POST) used and the parameters provided. Most parsing is left to the .NET framework, but some intelligence remains and the major task is the re-packaging of parameters for the device gateway core. Registration in IIS is simply done by creating a service URL as base request handler and IIS passes all requests down to the registered handler. This process is quite effective so high loads can be consumed as well.

Both variations (SOAP and REST) can be distributed in clustered environments as well as IIS is fully capable of clustered operations with load-balancing, etc.). From the side of any client using these communication interfaces no state is assumed and each operation is considered atomic.

5.9 Reflection on experience of creating reference implementation

Implementing the reference implementation using well known .NET technologies was straightforward without any major obstacles or problems, mainly due to the well-known behavior and interaction of the components. Considering an implementation in Java (or any other platform) should be no major challenge, given that the platform supports “active” components.

Integrating Python (as IronPython for the .NET framework) was a more complex task as passing data back and forth in a dynamic language environment (which was quite new at that time for .NET) posed some unexpected challenges. The same applied for the Ruby-integration (which was later abandoned due to lack of support for the ported runtime).

A very interesting area of the implementation was the integration of the F# runtime environment as it provides a whole range of options. Due to the functional programming nature of the environment (with all built-in benefits like the orientation towards parallel processing enablement due to immutable objects, and so on) this

could be a worthwhile area of research for more complex rules and data analysis. In general, this advanced "on-the-fly" analysis of data streams would be an important issue for later systems and versions. Especially with newer sensor types generating more data – and even data streams in a rapid succession – this poses different challenges than the current "point in time" analysis where a point is at the most compared to previous measurements and perhaps some additional data.

One major finding, which will have to be incorporated into next versions of the reference architecture, was that asynchronous operations have to be used to an even greater extent than they are currently used. A typical example would be the calculation of virtual sensor values as a result of an incoming data change. Currently it is a usable option to do this calculation synchronous to the inbound event, which is only usable as long as some threshold X is not exceeded. After that threshold X race conditions start to arise (new inbound events for changes might overtake previous calculations and thus data can be corrupted).

Stability, performance and extendibility had to be addressed by the reference implementation as well as, they are of paramount importance in using an architecture in a real environment / project.

Having undertaken the reference implementation, the next task was to evaluate the stability of the implementation, for example maximum number of sensors the system can handle, typical timings for accepting value changes, time needed to calculate virtual sensor values, and so on. Such measures are crucial for any integration where these factors have to be taken into consideration.

To address these questions in combination with a test of various requirements/characteristics from section 3.5 , several evaluation experiments were defined upon the reference implementation, and the results of these are detailed next in chapter 6.

6 Research Experiments

To establish basic performance characteristics of the reference implementation of the proposed Device-Business-Gateway Architecture (DBGA), as well as to evaluate if the requirements have been fulfilled, 10 experimental tests were undertaken with the reference implementation described in chapter 5.

Section 6.1 describes the test equipment used; sections 6.2 to 6.11 describe the tests performed; and sections 6.12 and 6.12 conclude with some overall observations.

For each test a description is presented under 5 headings:

1) The research question which was to be answered;
2) A description which explains how the test operates;
3) A definition as to which test parameters were used (and why);
4) A result section where the actual findings / measurements are detailed;
5) An analysis where the results are interpreted.

The overview in Table 6.1 shows which tests were conducted and which requirement is tested with it. In addition, it is shown which area (implementation or design concepts) is covered by the test.

Table 6.1: Experiment overview

Test #	Short description	Reason	Requirement covered	Area tested (implementation / design)
1	Timing to push values into the core with dynamic calculations	For the requirements “integrated data quality control” and “preservation of sensor state” timing considerations are of paramount importance to define usable limits of the implementation and technologies used	R6, R15	Implementation
2	Timing to calculate virtual values in the core using only internal data (available measures)	As “semantic data value enrichment” is one of the core features of DBGA it is important to understand limitations and implications of different approaches	R7	Implementation
3	Concurrent access (READ / WRITE) by several clients to check for concurrency, race conditions and locking issues	A general usability test to guarantee real-world operations		Implementation
4	Test with different data formats and conversion vs. native storage / handling	This test is for the requirement “data format agnostic”	R5	Design Concept
5	Core timing considerations when using workflows (triggering, execution control for long-running tasks)	This test is for the requirement “autonomous operation / workflow operation”	R8	Implementation
6	Communication mode (REST, WCF, .NET Remoting (Binary)) implications	This test is for the requirement “communication interface agnostic”	R4	Implementation
7	ODATA access	This is a combined test of the requirements: “centralized data store”, “direct data retrieval from data store” and “extendable and easy to change” as ODATA accesses the data in the centralized data store directly, yet is an extension to the core access patterns	R1, R2, R17	Implementation
8	Writing actuator values (by POSTing to a URL as a receiver)	This test is for the requirement “actuator support”	R9	Design Concept

9	MS-MQ adapter to READ / WRITE data with sample consumer / producer to prove the extendibility	This test is for the requirements “extendable and easy to change” as well as “provide feedback on data change”	R17, R11	Design Concept
10	MS-MQ adapter performance to evaluate usage in business workflow environments	This test is a by-product of the test #9 and useful as MS-MQ is used in many environments as a simple to use message queue system so performance considerations when using it together with the DBGA are important	R17	Implementation

6.1 Test equipment used

Tests #1-4 were performed on a standalone Intel i7 system with 2.9 Ghz and 8 GB of RAM and 256 GB SSD hard disk storage. In the "re-run" as well as for tests from #5 onwards a i7 8-core with 3.5 Ghz and 32 GB of RAM as well as 1 TB SSD was used

For test #6 a separate dedicated server with 2 Xeon 6-core-CPUs and 16 GB RAM as well as 4 TB RAID 1 of storage was used.

Both systems used Windows 7, 64 Bit and .NET 4.6.1 as a basis. The server was running Windows Server 2008 R2.

6.2 Test #1: Timing to push values into the core with dynamic calculations

Research question:

Each time a value is pushed into the gateway from an external source (for example sensor) some dynamic calculation (for example data filtering / data change) can take place before the value is stored internally. This causes some delay in the response (if called synchronously) as well as internal processing time.

The question is whether a given amount of x calculations / computations will cause significant (observable) impact to the whole system. As a side-effect a limit (upper-bound) for the number can be obtained.

Another important factor was which runtime environment for the computation (compiled, interpreted, or even the SQL database) can yield which performance given the amount of workload.

Description:

To answer the question a sensor variable was set (pushed to the gateway) 10,000 times and each time a calculation to perform a general scaling of the value (multiply by 4), before storing it internally, was executed. It does not matter what the actual calculation does, as long as it is always the same operation (to be able to better compare results) and that each iteration takes the same amount of time (which should be given if the same code is executed).

The calculation was implemented in C#, Python, SQL Server Stored Procedure and SQL statement (dynamically executed).

To allow for a swing-in and setup phase 200 iterations of the test were performed in a loop producing individual timings for each run so that MEAN, MIN, MAX and MEDIAN could be established (where MEAN and MEDIAN are usually roughly the same).

As the communication media can be quite important in the overall load and timing (see section 6.7 for examples) the test was performed with the value engine directly without any data protocol overhead.

Test parameters:

As the individual write is extremely fast 10,000 writes per cycle was chosen by testing to produce measurable as well as stable results.

200 cycles were identified (by using different numbers as well) as a reasonable small enough number to produce results fast (not too many unnecessary iterations) as well as large enough to get the results with only minimal test setup influence (right now around 3-4%).

Results:

Table 6.2 shows the results for one cycle with 10,000 executions for each type. All timings are given in µsec:

Table 6.2: Test 1 results in µsec for 10,000 executions

	MEAN	MIN	MAX	MEDIAN
C#	34,997	33,745	49,298	34,449
Python	65,897	62,390	90,059	65,518
Stored Procedure	753,890	751,166	772,842	757,366
SQL dynamic	1,266,839	1,247,701	1,291,689	1,265,811

The actual data for the results as a distribution chart:

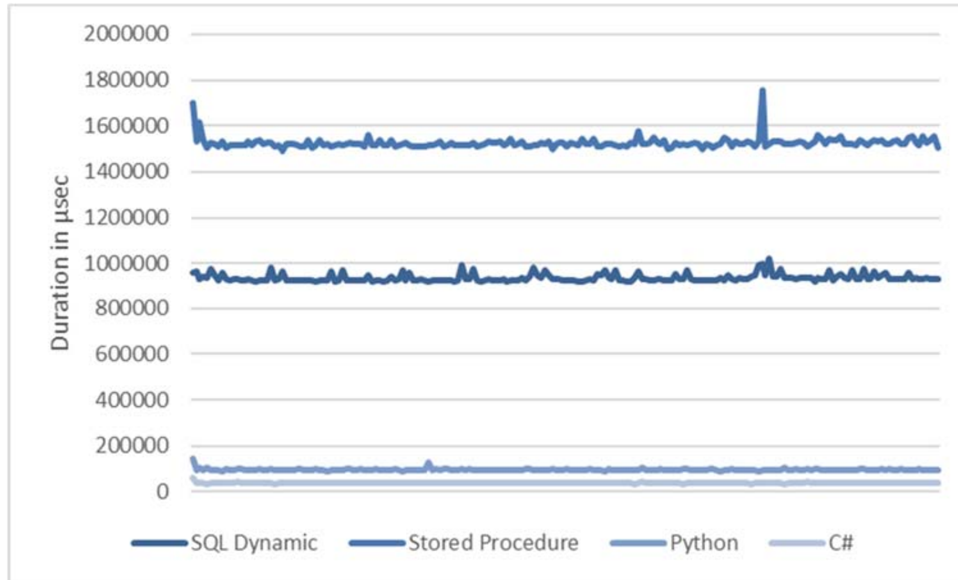

Figure 6.1: Test 1 results in µsec for 10,000 executions

The results show that evaluating 10,000 dynamic computations – including all overhead for retrieval, method resolution, dynamic dispatching, and so on – in C# cost between 34 to 49 msec (thus roughly 3 to 5% of the total available CPU time).

A specialized version of the test with 1,000,000 iterations showed no difference in linearity for C# but was omitted for the other environments.

Analysis:

The results were as expected showing that compiled C# is clearly the fastest and SQL dynamic execution the slowest operation. Yet the interesting point was that Python, being purely interpreted at runtime, still is only 2 times slower than C# and thus absolutely capable of being used for triggers as well.

To put it differently: if 10,000 sensor values are pushed into the core and each value is evaluated, checked and some simple computation implemented in dynamic Python is undertaken (like the sample calculation or a simple band-pass filter), then the total execution time is 90 msec as the worst case. The usage of a stored procedure could be

acceptable, depending on the overall load of the system, but has to be evaluated in a concrete case.

Given that the system utilization for C# is between 3 and 5% and for Python between 6 and 9%, this implies a theoretical upper limit of roughly 10 times the amount of calculations in Python and 20 times in C# resulting in theoretically 100,000 value settings and calculations / s in Python and 200,000 in C#.

Considering an environment with 200 sensors (which already is quite big) and each sensor fires once every 10 msec (therefore 100 times / s, which is very fast as well), the total data changes would be 20,000 and thus well within the boundaries of the system.

With dynamic calculations being a parallelizable algorithm, a future version of the implementation should utilize multi-core CPU environments specifically. In addition, by means of independent calculation tasks, the current synchronous operation could be split, thus making the system more responsive and able to handle even higher data loads.

6.3 Test #2: Timing to calculate virtual values in the core using only internal data (available measures)

Research question:

As a core feature of the Device-Business-Gateway Architecture (DBGA) is the enrichment of data by means of virtual sensor data, it is important to know how much time is consumed in such calculations and therefore what would be the upper limit of requests which can be handled in a given time X.

To make sure that no external factors (like network latency for example) interfere with the result, only internal data was taken as a data basis for the calculation.

Description:

The virtual value calculation was evaluated with an implementation in C# and Python (SQL Stored Procedure would be future work).

In the calculation all defined historic values (buffer size was 10 values) were added up dynamically and the result (as a SUM() function) was delivered. Given this, the operations were the same for each iteration and each execution was the same length as the previous one.

The test involved not only the calculation (and value conversions from internal to numeric representation to perform the summing) but the access to internal value structures as well (with all boundary checks, and so on.).

As the communication media can be quite important in the overall load and timing (see section 6.7 for examples) the test was performed with the value engine directly without any data protocol overhead.

To allow for a swing-in and setup phase, 200 iterations of the test were performed with 10,000 iterations of the evaluation, in a loop producing individual timings for each run so that MEAN, MIN, MAX and MEDIAN could be established.

Test parameters:

The same test parameter criteria applied as in test #1 (section 6.2).

Results:

Table 6.3 shows the results for 10,000 executions for each type. All timings are given in µsec:

Table 6.3: Test 2 results in µsec for 10,000 executions

	MEAN	MIN	MAX	MEDIAN
C#	99,276	97,464	131,640	98,846
Python	219,655	213,768	248,721	218,722

The actual data for the results as a distribution chart:

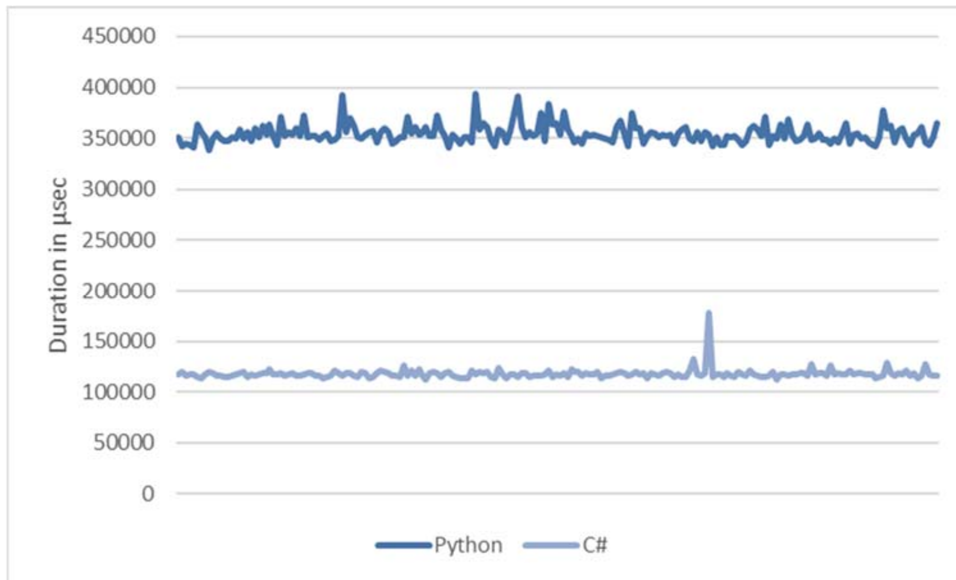

Figure 6.2: Test 2 results in µsec for 10,000 executions

Again, C# is roughly 2 times faster than Python, but still Python is fast enough to deliver 10,000 values in 220 msec.

Analysis:

Given the results, it is obvious that evaluating virtual sensors is a time-consuming task which should not be underestimated in terms of impact on the whole gateway. As the execution currently is synchronous on behalf of the request (with parallel requests executing the virtual sensors in parallel as well) the response time for a request can be delayed if load increases.

Yet if 100 virtual sensors would be queried every 100 msec (which already would be an extremely high value as a refresh rate of 100 msec is rarely used) only 1,000 requests would be made, occupying the system around 1-2% (depending on the implementation).

As long as the implementation of the virtual value uses only internal data, timings will always be in a linear range as everything is loaded in the cache and thus no delays for out of bounds access, etc. has to be considered. In such a scenario, the main issue would be the conversion routines as they usually consume quite a lot of CPU cycles. As virtual value requests could cascade if the virtual value depends on other virtual values, the whole execution path could become rather complex.

In case external data (by means of for example web service calls) is accessed to evaluate the virtual value, timings can no longer be predicted.

Due to the fact that requests for virtual values are usually initiated from a request from a client and requests from all clients are handled as individual parallel units of work (by means of the communication handler, for example ASP.NET) the mutual influence would be lessened.

6.4 Test #3: Concurrent access (READ / WRITE) by several clients to check for concurrency, race conditions and locking issues

Research question:

Mixed read and write data access by several clients to the same data item usually produces issues in several areas. Therefore, it is of paramount importance for the DBGA to gain insight into concurrency, locking and race conditions when parallel access to the same data cells is performed.

To bring the test more to a real-life scenario writing a new value shall involve triggering a calculation as well, so that threading issues show up better.

Description:

For this test 5 reading and 5 writing threads concurrently accessed sensor values with each thread performing 500 write and read operations per iteration – yielding 10 parallel requests in the gateway at any point of time. The total load therefore was 2,500 read and 2,500 write operations per iteration over all threads.

The writing thread had a trigger on its value (again a simple multiply by 3) and the read thread was directly accessing the values as such. The trigger was implemented in C#, Python, SQL Stored Procedure and dynamic SQL.

To allow for a swing-in and setup phase 100 iterations of the test were performed in a loop.

Test parameters:

As 10 parallel requests at any given moment relate to 10 concurrent sustained data consumers / producers at all times, this was considered a realistic real-time environment as very seldom more than 10 parallel operations should take place at the very same moment[131].

By using 500 operations for each thread in one iteration the total time used is large enough to be measurable and small enough to be quickly available.

Results:

[131] The test was run with 200 parallel operations for C# alone (so 20 times the amount of concurrency) and the execution time just 2,5 times longer giving a good scaling for the parallel access

Table 6.4 shows the results for one iteration (having 5 executed in parallel) using 500 operations per iteration. All timings are given in µsec:

Table 6.4: Test 3 results in µsec for 5 parallel read and 5 parallel write executions (500 times each)

	MEAN	MIN	MAX	MEDIAN
C# read	808	133	31,572	301
C# write	2,831	427	38,039	1,068
Python read	880	134	31,141	300
Python write	10,854	1,087	65,935	6,975
SQL dynamic read	2,364	456	45,786	1,026
SQL stored procedure read	1,874	438	40,846	970
SQL dynamic write	1,307,811	998,488	1,644,265	1,299,924
SQL stored procedure write	1,204,606	876,726	1,538,354	1,245,710

The actual data for the results as a distribution chart for reading operations:

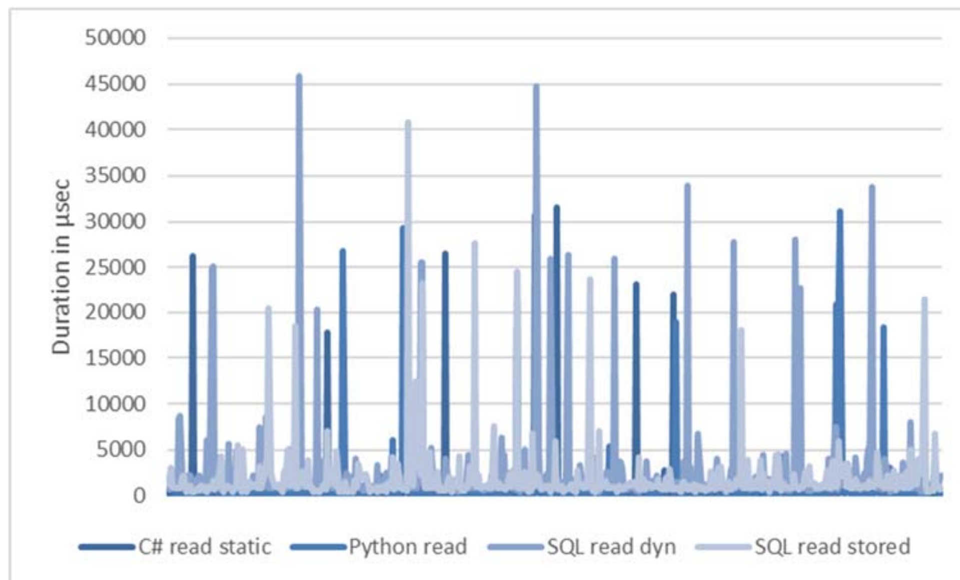

Figure 6.3: Test 3 results in µsec for 5 parallel read and 5 parallel write executions (500 times each) – reading operations

The actual data for the results as a distribution chart for writing operations:

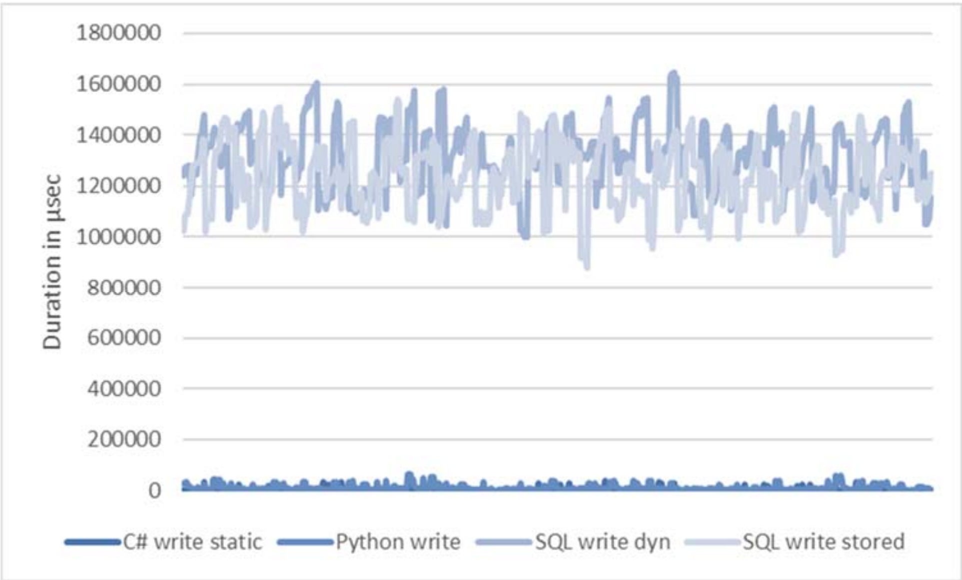

Figure 6.4: Test 3 results in µsec for 5 parallel read and 5 parallel write executions (500 times each) – writing operations

Each "read" indicates the reading of the value of a sensor, each "write" is the setting of the value inclusive of the trigger execution in the corresponding environment.

Analysis:

Again, C# is performing the fastest, yet interestingly this time the performance against Python is not 2 times, but 4-6 times better, which might be mainly attributed to faster thread switching and synchronization issues. SQL performance is in line with previous tests and no anomalies are showing up.

In general, it can be concluded that parallel access from several execution threads with intermixed read and write works fine and is very stable. For larger scenarios SQL should not be used as an environment where triggers are implemented as the total execution time might simply be affected too much.

It is interesting to compare the C# results with the results of test #1 (section 6.2). In test #1 the MEAN for 10,000 write operations (using triggers) is 34,997 µsec which relates nicely to the MEAN timing for 10,000 write operations (20 * 2,872 = 57,440 µsec). In the current test the read operations are "on top" as well as the thread access synchronization. In a mixed 10,000 operations this test would produce 10 * 2,872 µsec for write and 10 * 807 µsec for read which yields 36,790 µsec, making it similar within 10% to test #1.

In a separate test (not included in the thesis) it might be worth investigating if a separation of the SQL Server engine to another system (and therefore removing of load from the system) produces better performance for the SQL use cases. This is especially interesting as triggers very often are based on historic data which might be still in the database or alternative data sources and therefore the closeness of the trigger evaluation to the data source could be enhancing the overall performance.

6.5 Test #4: Test with different data formats and conversion vs. native storage / handling

Research question:

As the proposed device gateway stores all values internally in a character representation in the database (data format agnostic - without any interpretation of the value as such) it was important to see if that causes any performance penalty.

Description:

To research the question a test was constructed which was implemented similarly in native T-SQL (MS SQL), C# and Iron Python (both using the SQL Server connection).

The test was divided in two parts:

The first part ran as a loop for 5,000 iterations and in each loop an operation was performed 1,000 times. In the operation a simple INSERT statements with dynamic (Dynamic…) and static value conversions (Fixed…) was executed, comparing the conversion from the internal format to a string and storing the internal format in a fitting native SQL type like BIGINT, INT, etc.

In the second part of the test a more “typical” operation was simulated – reading integer and decimal values from a table, adding something and writing them back (as INSERT, not UPDATE). The internal values were converted in the data receiver from the string format (DynamicTransform…) into the native data types or used the “correct” native SQL server types (FixedTransform...) to retrieve data. As a special case the raw SQL performance to perform these INSERT operations using pure SQL was measured as well.

The second part was iterated 5,000 times as well and each sub-part read and wrote 1,000 values to generate more load and not use the cache all the time.

Test parameters:

5,000 iterations were chosen to yield a stable measurement as with smaller numbers the test setup was too noticeable. With 1,000 operations per iteration the tests produced sufficiently usable data without too much setup influence.

Results:

Table 6.5 shows the average timing for 1,000 conversion operations in μsec either using a dynamic conversion (string -> native data type) or a fixed conversion (reading directly from a column with the same data type).

Table 6.5: Data format conversion performance of SQL, Python and C# using native and agnostic data formats (in μsec)

Sub-Test	SQL	Iron Python	C#
DynamicBigInt	58.84	127.64	122.54
DynamicBit	58.74	127.68	122.66
DynamicFloat	59.72	128.5	124.22
DynamicInt	58.26	127.72	122.6
DynamicNumeric	60.28	129.88	124.7
FixedBigInt	58.14	120.92	116.86
FixedBit	58.2	120.5	116.54
FixedFloat	58.66	121.1	116.74
FixedInt	58.66	120.94	120.52
FixedNumeric	58.88	121.46	117.64
FixedString	67.46	129.08	125.34

Table 6.6 shows the average timing for 1,000 conversion operations in μsec either using a dynamic conversion (string -> native data type -> string) or a fixed conversion (reading / writing directly from / to a column with the same data type).

Table 6.6: Data format conversion performance of SQL, Python and C# using native and agnostic data formats (in μsec)

Sub-Test	SQL	Iron Python	C#
DynamicTransformInt	63.22	138.86	128.48
DynamicTransformNumeric	64.79	142.97	131.19
FixedTransformInt	63.17	133.65	122.80
FixedTransformNumeric	64.36	135.91	124.11
SelectBasedFixedTransformInt	3.22	3.34	2.80
SelectBasedFixedTransformNumeric	3.17	3.50	3.00

In the Fixed variants the value was read directly into a variable of the same native type (an SQL-INT into a 32-bit int and a NUMERIC into a decimal object), whereas for the

Dynamic variants the data was read in a string version, converted to the target format, then processed and for the writing converted back to a string-version. Thus the Dynamic variants involve 2 convert operations (one for retrieving, one for writing).

The SelectBased… variants timing is for a comparison with native SQL performance.

Analysis:

As can be seen differences among the different data formats and the conversion (the Dynamic variants) are not really existing which proves that storing the data in a non-native format is of no concern. This applies for reading as well as writing.

Another general result this test provided is the closeness in terms of performance between C# and IronPython. Technically both are interpreted languages, yet the primary intutition would have been that C# would be much faster than Iron Python. Yet as the results show there is just some 10% speed increase which means in terms of data processing both approaches should be approximately similar. Given this observation, it can be considered that Iron Python could be a first class implementation environment for triggers and filters, since these have to operate as fast as possible.

Due to the fact that the SQL engine can keep all data internally (and not transport over an external protocol), the pure SQL operations are generally far superior, which clearly indicates that for heavy duty operations pure SQL should always be favored (for example large scale calculations, etc.).

This insight is highly valuable as all operations which require mass data handling will therefore be optimized towards SET-operations wherever possible in the gateway.

Size-wise the string-representation takes up more space in terms of hard-disk usage, of course. Yet this is not very much more, as a float would anyhow require 6 bytes internally and the average string representation might be done in 10 - 14 bytes. Given the current disk capacities and the cost of storage this point can be ignored in the further discussion except if trillions of data items are to be stored.

6.6 Test #5: Core timing considerations when using workflows (triggering, execution control for long-running tasks)

Research question:

Due to the importance of workflows for the DBGA, the performance evaluation of these in virtual sensor evaluation and trigger execution compared to other options is necessary.

Description:

The test cases in section 6.2, 6.3 and 6.4 were extended with a workflow running as a code activity (which is a code only Windows Workflow item performing some operation). This code activity is used for a trigger (one for general checking, one as a trigger after the value changed), for a virtual value calculation and for the read / write parallel operations.

The code activity performs exactly the same operation as its C# or Python counterpart.

Test parameters:

The same test parameters as in section 6.2, 6.3 and 6.4 were applied.

Results:

The three different test case adaptions can be seen in the following results:

- Table 6.7 shows the workflow acting as a trigger for the general check (data quality)

- Table 6.8 shows the workflow performing a virtual value computation

- Table 6.9 shows the adapted parallel read / write scenario of section 6.4

The lines for the other results are simply presented as a comparison to the workflow's performance.

Trigger implementation

Table 6.7: Workflow as trigger for checking and after change (in µsec)

	MEAN	MIN	MAX	MEDIAN
C#	34,997	33,745	49,298	34,449
Python	65,897	62,390	90,059	65,518
Stored Procedure	753,890	751,166	772,842	757,366
SQL dynamic	1,266,839	1,247,701	1,291,689	1,265,811
Workflow Check Trig.	84,945	82,941	123,568	84,506

Virtual Value Calculation

Table 6.8: Workflow as virtual value calculation (in µsec)

	MEAN	MIN	MAX	MEDIAN
C#	99,276	97,464	131,640	98,846
Python	219,655	213,768	248,721	218,722
Workflow	211,356	208,800	262,825	211,012

Read / Write parallel operation

Table 6.9: Workflow in a parallel read / write scenario (in µsec)

	MEAN	MIN	MAX	MEDIAN
C# read	807	133	31,572	301
C# write	2,872	427	38,039	1,071
Python read	880	134	31,141	300
Python write	10,924	1,087	65,935	6,978
SQL dynamic read	2,358	456	45,786	1,026
SQL stored procedure read	1,872	438	40,846	972
SQL dynamic write	1,307,599	998,488	1,644,265	1,298,785
SQL Stored Proc Write	1,238,865	876,726	1,538,354	1,243,732
Workflow Read	1001	404	14,349	421
Workflow Write	11,322	2,312	33,121	7,121

Analysis:

The workflow implementation is – as expected – slightly less performant than Python and C#, yet still very competitive. Especially as workflows as triggers and virtual value evaluations involve a lot of overhead, with loading the agent definition, initializing the environment, and so on.

Due to their abstract and declarative nature (they can be visually constructed in the Windows Workflows environment using Visual Studio) and their extended capabilities being available for use in real-world environments is a big benefit. This test proves that they are perfectly usable and capable to fulfil the needed functionality when considering the timing and load issues involved.

Further tests (outside the scope of this thesis) have to show if a separation of these agents to a different system (activation can be done remotely as well, even via web-service activation calls) would cause a significant timing difference. This is anticipated as the switching of the environments causes quite some load issues on the system which would be eased.

6.7 Test #6: Communication mode (REST, WCF using SOAP, .NET Remoting (Binary)) implications

Research question:

As the DBGA is supposed to be communication interface agnostic it is important to know which execution parameters that the communication interface has. As the main protocols are REST, SOAP and .NET Remoting (as a binary protocol) these have to be investigated accordingly.

Description:

The same test was run 3 times – each with a different protocol: REST, SOAP and .NET Remoting (binary). In each run the test process was launched 20 times in parallel and 10 values were retrieved for 50 times in 1, 5 and 10 threads in parallel. This resulted in 20, 100 and 200 requests in parallel (20 processes with 10 threads each making a request) and a total of 10,000 requests (50 requests for 200 threads) simulating a very large number of users as these values in such a short time would be quite unusual for a normal installation.

To allow for a swing-in and setup phase of the system, which was especially important for the larger thread counts, the first 1, 4 or 6 rows of result data (thus accounting for 1, 5 and 10 parallel threads) were removed from the measurements as these are clearly exceptional results due to setup phase.

The values retrieved had no calculated data and were served directly from the internal cache so no additional computational time was used which is a typical use case as well. Very often data will be coming from “current” values and as this test is supposed to test for the communication protocol there is no need for further elaboration of this part.

Test parameters:

The thread-count per process was not increased beyond 10 as then inter-thread problems started to show up[132], which have nothing to do with the device gateway, but more how the communication channels are used and dispatched by Windows per process. In addition, 10 threads retrieving values in parallel from one process is rather uncommon in real-world scenarios, where mostly 1-3 or 4 reading threads would be used to scan for data changes (and even then only every 5 seconds or less as this would be busy polling otherwise).

To use 20 processes simulates parallel users as then each process could be associated with a user. To use more processes is counter-productive as Windows starts to use more and more time for process dispatching and less for actual processing. Should a larger user number be relevant, then the test must be performed using different systems for clients and server.

Results:

To provide realistic timings the values were averaged over all 20 processes and given in μsec for a single iteration (50 requests).

Table 6.10 shows the results for 1, 5 and 10 threads in parallel for .NET Remoting, SOAP and REST as protocols used.

Table 6.10: 1, 5 and 10 threads in parallel retrieving data using .NET Remoting, SOAP and REST as protocol (in μsec)

	MEAN	MIN	MAX	MEDIAN
REM 1	4,234	345	23,976	2,255
REM 5	26,471	418	171,551	14,801
REM 10	38,714	372	528,262	24,602
SOAP 1	171,723	6,302	381,295	161,919
SOAP 5	122,604	4,971	500,971	98,328
SOAP 10	100,174	7,387	511,261	81,452
REST 1	9,656	5,061	20,385	8,679
REST 5	79,747	11,914	362,287	64,149
REST 10	110,232	13,199	431,001	102,012

[132] Especially SOAP had problems with more than 20 threads and stalling issues

2 machine execution:

Table 6.11: 1, 5 and 10 threads in parallel retrieving data using .NET Remoting, SOAP and REST as protocol between 2 machines (in µsec)

	MEAN	MIN	MAX	MEDIAN
REM 1	7,301	360	31,969	4,792
REM 5	39,200	422	211,076	24,283
REM 10	40,688	355	301,696	20,142
SOAP 1	40,263	6,558	174,920	35,324
SOAP 5	123,131	13,046	1,455,249	63,078
SOAP 10	207,221	8,573	3,209,347	131,205
REST 1	23,383	2,761	113,708	21,724
REST 5	89,595	12,165	428,354	72,338
REST 10	60,107	3,658	234,039	56,003

The actual data for Remoting as a distribution chart:

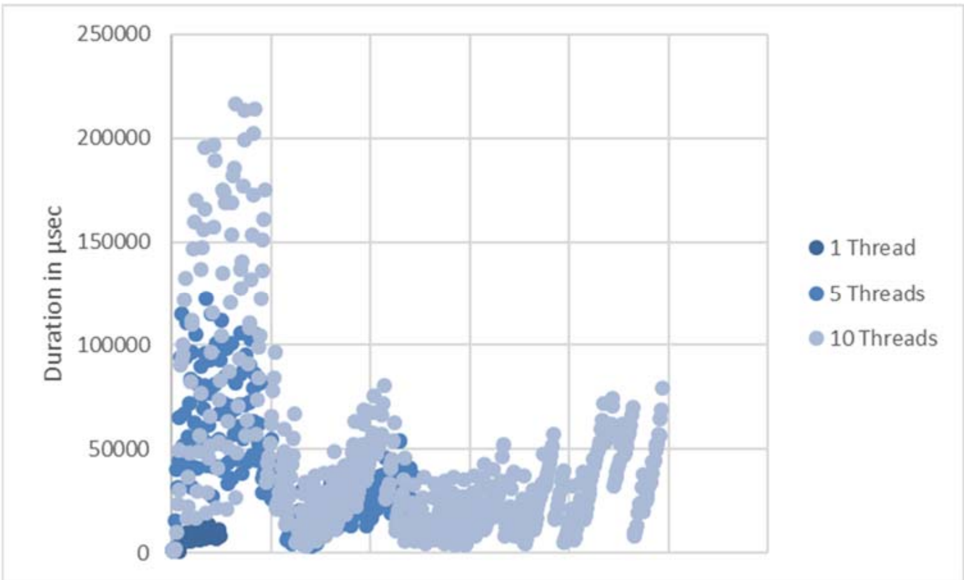

Figure 6.5: 1, 5 and 10 threads in parallel retrieving data using .NET Remoting as protocol (in µsec)

The actual data for SOAP as a distribution chart:

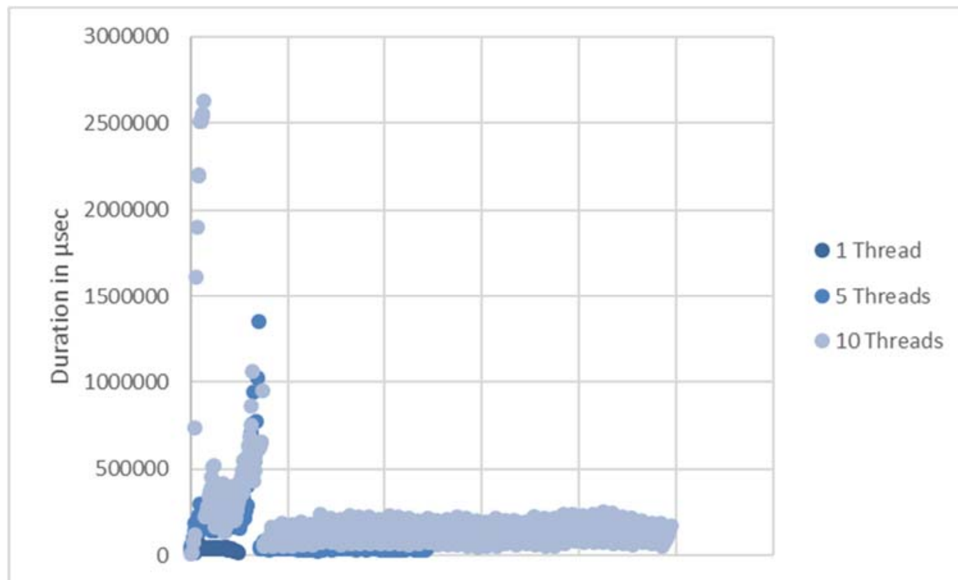

Figure 6.6: 1, 5 and 10 threads in parallel retrieving data using SOAP as protocol (in µsec)

The actual data for REST as a distribution chart:

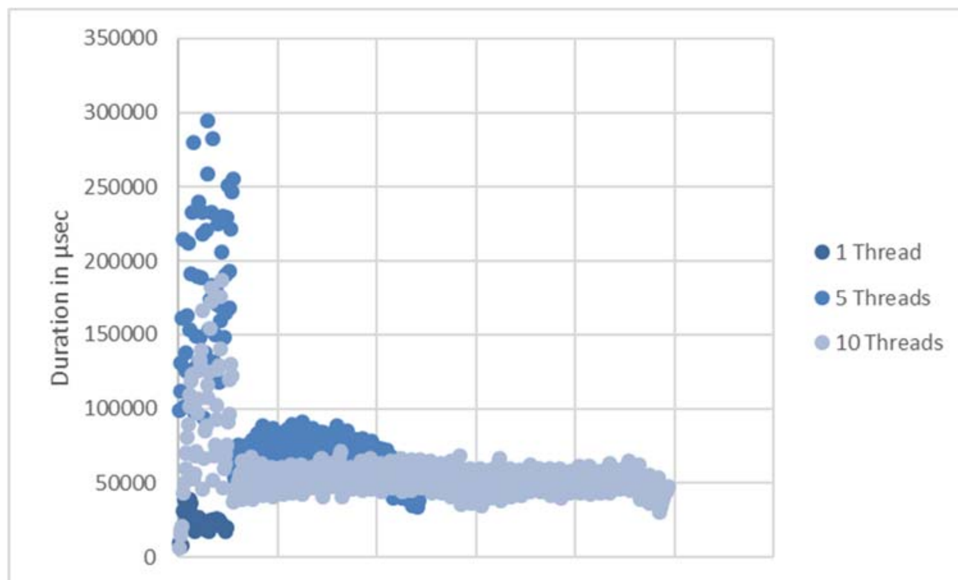

Figure 6.7: 1, 5 and 10 threads in parallel retrieving data using REST as protocol (in µsec)

It should be noted, that in the reference implementation the protocol handlers are a façade to the core, internally .NET Remoting as a protocol between for example the REST handler and the core is used. Therefore, the REST-timing includes the .NET Remoting timing for the request as well (including parameter conversion, etc.). WCF has some setup advances over REST as resources are reused which minimizes setup time as (compared to REST where the setup has to be done every time).

Analysis:

As setup costs for communication links are quite high, they have to be considered for single requests as these setup costs will contribute quite a lot to the total request time.

To give an example in the single threaded REST request the 1st request iteration (which was eliminated from the above result as a pure setup exception) takes 1,090,410 µsec (so more than 1 sec) whereas subsequent ones have an average of 7,301 µsec. This is mainly due to establishing the IP link, opening ports, HTTP setup, and so on. This shows clearly that for efficient operation the client should try to maintain connections open / connected as long as possible. In case the route, and so on, has to be established for each new request, the global overhead will be considerable.

The results show clearly that the system performs quite nicely when for example 50 value requests using REST are served in a medium load environment (5 threads in parallel on 20 processes = 100 requests in parallel) in 79 msec as average.

Another – quite surprising – result was that REST is doing so well in comparison to the binary protocol used by .NET Remoting by being just three to five times slower. This is even without considering caching, which might be an option for often queried values that do not change often. Therefore, REST can be considered a real promising, fast and good integration solution. Another observation was that the overall system load using REST was at roughly 55% CPU utilization whereas SOAP drove the system to 100% and stalling.

SOAP being the most verbose of all tested options (with request / response packet sizes being 430 bytes vs. 125 bytes for REST) clearly used the most time and system resources. When running the tests on a single machine, the system was stalled.

In future stages (not part of this thesis) it has to be evaluated if the time consumption is mainly driven by the protocol handling (parsing) or the WCF method resolution / routing engine (which uses a lot of reflection for parameter passing, etc.).

Considering these results SOAP is a viable option for integration, yet it is much slower than for example REST and therefore the data transmission rates or request rates have to be adjusted accordingly. Especially for small single data requests (for example the latest sensor value) the time needed is far higher than REST (nearly 18 times slower).

Interestingly when running the test in a dedicated server environment timing change considerably with .NET Remoting becoming 4 times slower and SOAP becoming 3-6 times faster, drawing almost equal with REST in most cases. The decrease for .NET Remoting was expected as the link shifts from a loopback TCP/IP to a “real” network

link, yet for SOAP in general and REST for the higher bandwidth scenarios, this was quite a surprise. It all points in the same direction, that setup costs are a considerable factor in the whole equation and between two machines load issues (concurrency, memory, etc.) faced by SOAP in a single environment, is alleviated. Therefore, SOAP and REST could be used interchangeably.

Another experiment for a later stage (not part of this thesis) could investigate if in a self-hosted environment (thus not using WCF for example for REST) execution time is faster due to for example simpler routing mechanisms.

Stalling and overload issues have to be considered in a real environment as timings have a wide spread between MEAN and MAX (up to 6 times) which makes execution timing quite unpredictable. In particular the MAX values can be unusual high due to communication setup and resource buildup reasons, yet with a real server (which would give priority to background processes like for instance the IIS in the DBGA case) this should not happen in a production environment.

6.8 Test 7: ODATA access

Research question:

ODATA[133] is an important protocol used by most business users who want to access and consume sensor data from applications like for example Excel. As it is mostly used to only consume (despite its capability to update and delete as well, which is not supported by the reference implementation), the important question was which limits are imposed and how does the architecture scale.

Description:

The ODATA service of the reference implementation is implemented as a WCF-service and derives from an EntityFramwork (EF)[134] data service, which makes the implementation straightforward (around 20 lines of code + 3 configuration files are

---

[133] Open Data Protocol (OData) is a data access protocol to provide standard CRUD access to a data source via a website. It is similar to JDBC and ODBC although OData is not limited to SQL databases (http://en.wikipedia.org/wiki/Odata)

[134] An open source object-relational mapping (ORM) framework for ADO.NET, part of .NET Framework to simplify SQL Server and other relational database access with designers, tools and runtime support

needed). The service provides the ability to any ODATA client to retrieve devices, sensors for devices and values for sensors.

Excel is accessing ODATA through the PowerPivot add-in which retrieves the data items and then stores them internally in Excel format – thus providing a disconnected facility – which is not interesting in terms of load measurement as the data is retrieved only once. Therefore, an ODATA test client has been written which by means of LINQ and ODATA accesses the ODATA sensor data feed and retrieves a configurable amount of data items in configurable steps.

Test parameters:

As test data 1,000,000 sensor data records were generated by a SQL script, so each query addresses the complete data set.

This data was queried from a remote client (as local client access will not happen in reality very often). In the first scenario a small amount of data (retrieving 100 – 9,900 sensor records with a step width of 100) and in the second a large amount of data (retrieving 5,000 – 195,000 sensor records with a step width of 5,000) was queried.

Having different retrieval rates is a useful comparison as a typical client will request smaller packets in a more frequent approach, while data mining applications will be interested in larger scale requests for more data. As 200,000 seemed enough sensor data for a normal consumer the tests were limited there.

Results:

Figure 6.8 and Figure 6.9 is for retrieving 100 – 9,900 records (step width 100) on different machines for client and server.

In Figure 6.8 the request processing time for the server is shown in msec

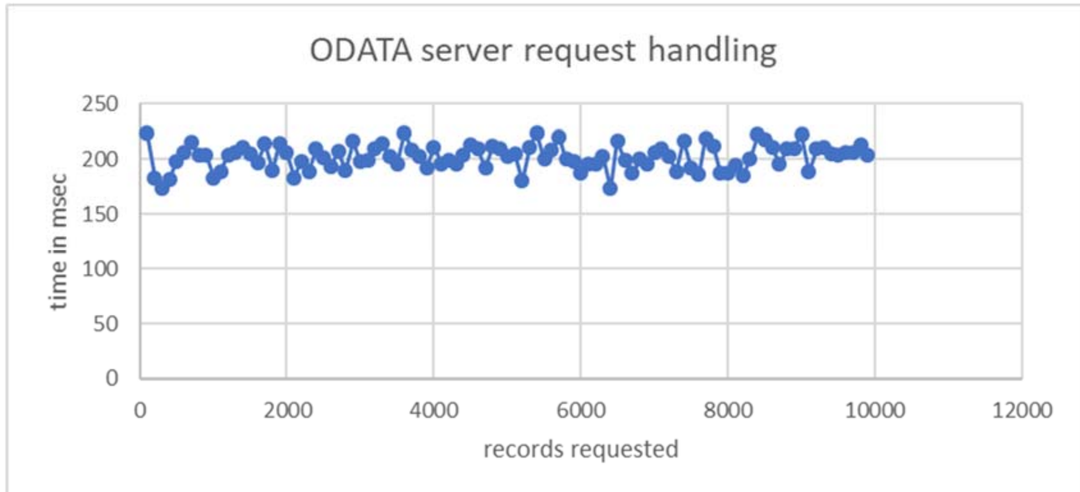

Figure 6.8: ODATA server processing time in msec for 100 - 9,900 records

In Figure 6.9 the total request time for the client is shown in msec

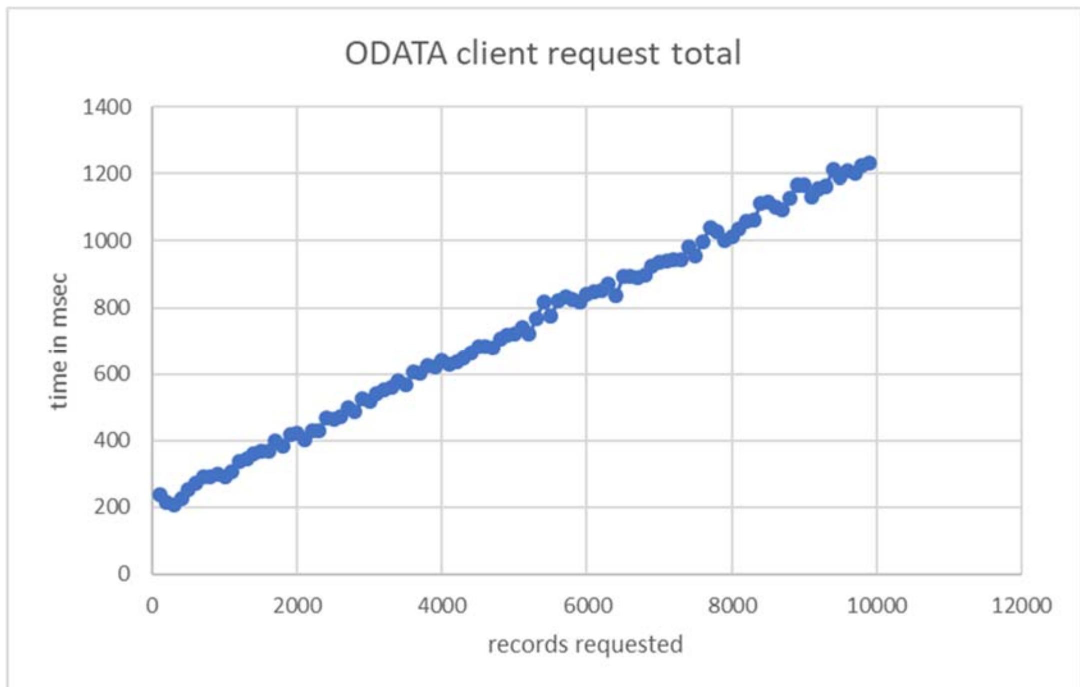

Figure 6.9: ODATA client request time in msec for 100 - 9,900 records

Figure 6.10 and Figure 6.11 is for retrieving 5,000 – 195,000 records (step width 5,000) on different machines for client and server.

In Figure 6.10 the request processing time for the server is shown in msec

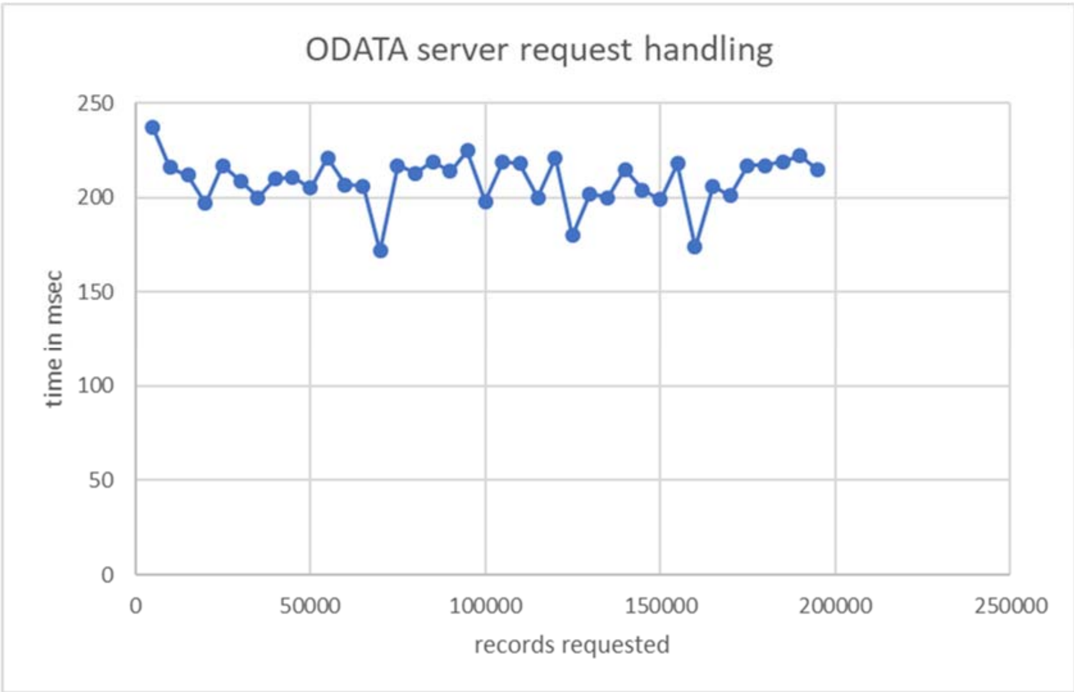

Figure 6.10: ODATA server processing time in msec for 5,000- 195,000 records

In Figure 6.11 the total request time for the client is shown in msec

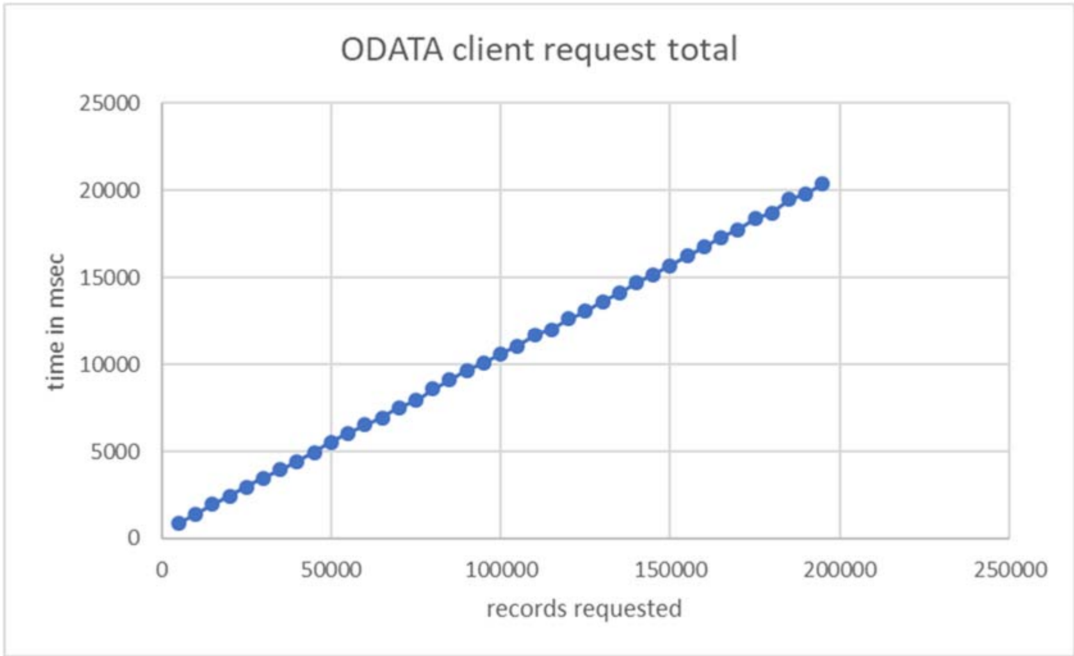

Figure 6.11: ODATA client request time in msec for 5,000- 195,000 records

Analysis:

The server processing timing shows an almost linear (+/- 20%) distribution of processing time mainly due to the fact that the data, after the initial swing-in-phase was dealt directly from the cache. In addition, regardless of the requested records the timing is approximately identical with only slightly higher values for the larger data sizes.

As all the data has to be converted on the server into XML (not measured in the server processing time) and sent back to the client where it is parsed to the internal format this contributes to the client timing. As can be seen the client timing is an almost linear function which is only dependent upon the size of data items – thus directly related to the size of the XML data packet.

Up to 8,000 records the total time is still under 1s which usually would be acceptable by users, according to the experience of the author.

While performing the tests it showed that the ODATA client had a problem around 510,000 records, because the XML serializer ran out of memory, which resulted in a reduction of the maximum record request size to 250,000. This has to be considered in real-life scenarios when server to server communication might be used as then higher record sized could occur. Yet ODATA as a protocol for such large data requests does not really seem to be appropriate and different retrieval mechanisms should be utilized.

6.9 Test #8: Writing actuator values

Research question:

Writing data to an actuator is a functionality which, especially if done synchronously to the request, needs very precise timings to be able to estimate how many requests can be handled by the architecture and how these will be distributed time-wise.

As usually one data item is written at one time, this is the most important timing. However, to have values for several writes in one point of time, gives insight how the value will increase based on the amount.

Description:

As the network connection speed can be very much dependent on the environment used this was not considered in the test so only a local installation was used. The issuing process, the reference implementation and a sensor process (acting as a receiver of the write) were on the same local system, connected by TCP/IP, thus testing the core internal performance of the gateway implementation.

Test parameters:

To check for different execution times, the test was performed using 10,000 requests with 1 sensor value per write request and as alternative with 500 requests and 20 sensor values per write request, thus the same total amount of data.

In total, using the 10,000 writes, execution timing was stable enough to exclude setup issues.

Results:

Figure 6.12 shows the total duration distribution in µsec of 10,000 requests with always 1 sensor value per write request.

In Figure 6.13 the same information can be found, just for 500 requests with 20 sensor values per write request. The big difference is that for the 20 sensor writes just one request from the client to the server is made, yet from the server to the device 20 updates have to be done.

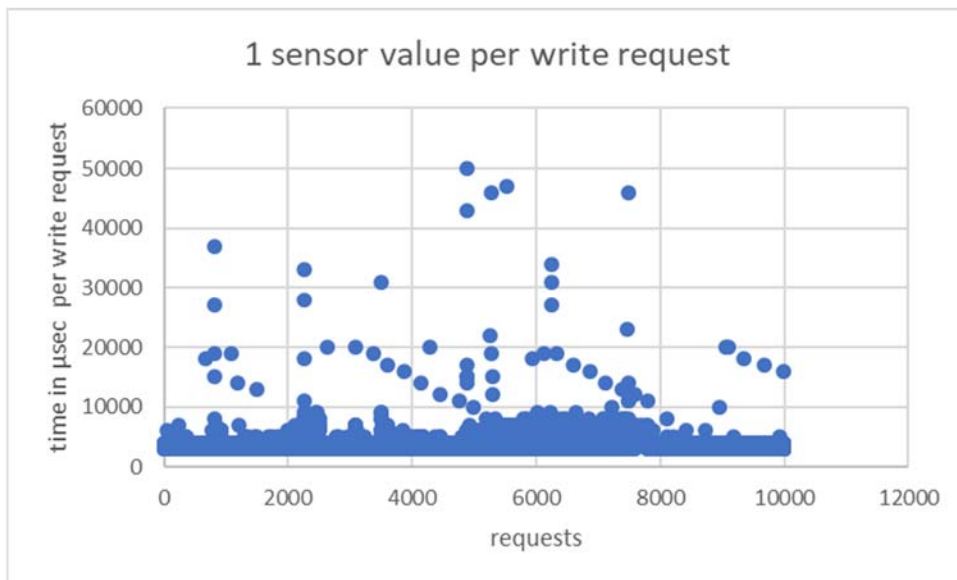

Figure 6.12: total time in µsec for writing 1 sensor value to an actuator

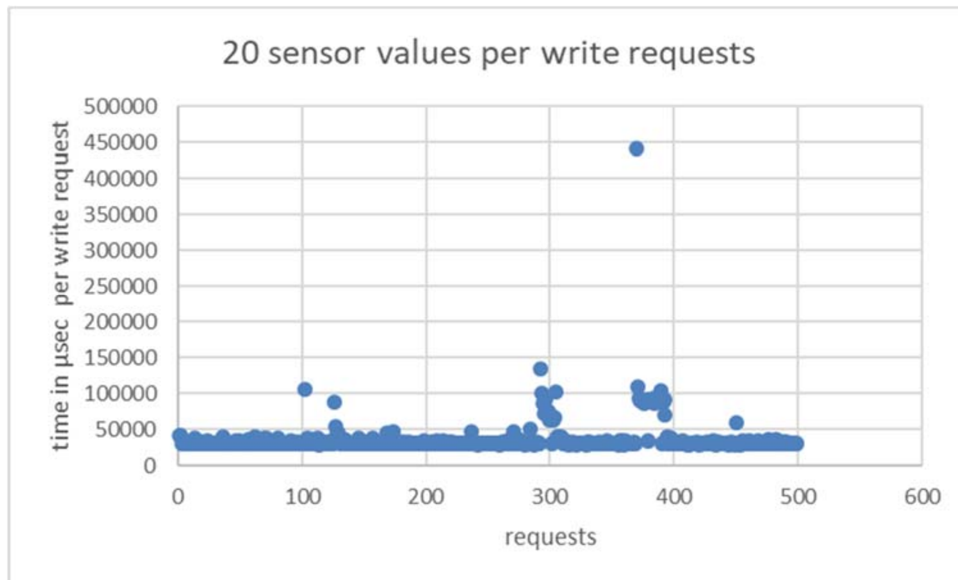

Figure 6.13: total time in µsec for writing 20 sensor values to an actuator

Analysis:

Initially the tests were undertaken using UDP as well, but it was discovered that UDP can only be used for approx. 7-8 sensor data items / request as otherwise the packet size (in JSON-format) exceeds 1500 bytes, which is the default MTU[135] of the network card. Any traffic larger than this value cannot be transported in one packet and the overhead to implement multi-packet UDP requests was as well too high, as against the idea of having "simple" packets without delivery guarantee as well.

Yet if the non-guaranteed nature of UDP can be accepted, and the request size does not exceed the MTU, then UDP might be a real option for remote scenarios. In the same test as above performed using two machines (client + actuator on one, server on the other) the average timing for TCP with 1 sensor value / request was 253,656 µsec compared to 87,722 µsec for UDP, but this could vary widely based on the network infrastructure, routers, etc.

One more issue which has to be researched in a later stage (not part of this thesis) would be how a connection caching in the service could enhance these timings. Currently whenever a new write request is issued the server will acquire a connection, pass the value and close the connection again (as otherwise the connections might run out), yet this connection could be maintained if several writes are supposed to occur to the same destination.

---

[135] MTU = Maximum Transmission Unit

6.10 Test #9: MS-MQ adapter to READ / WRITE data with sample consumer / producer to prove the extendibility

Research question:

Extendibility is a major feature of the DBGA so it is important to know how easily and straightforwardly such an extension could be done. As messaging architectures are used by other gateway systems as well, having an MS-MQ adapter as a special extension providing a message queuing implementation to the available communication protocols is of special interest, too.

From the several available message queuing options like MS-MQ (which comes with the Windows operating system), BizTalk (an independent product from Microsoft), IBM's MQ Series, Rabbit MQ[136] and various others, MS-MQ was chosen as it is free of charge and directly integrated in Windows.

Description:

In the test, a client writes sensor values to an actuator via the gateway using MS-MQ for all communications between client, server and actuator. The data to send to the actuator is placed as a XML- message in the server's in-queue where the server retrieves the message, processes it and a response is written to the server's out-queue, where it is read again by the client (so a kind of synchronous operation takes place with send – process – acknowledge).

The actuator is informed by placing a XML-message into the server's actuator out-queue where it can be read by any client.

For the test, the server was extended using a MS-MQ handling facility (plug-in) and the actuator notification was extended with another communication handler to use MS-MQ (aside HTTP REST). The queue names used are configurable and thus a scaling out to any infrastructure is simply a matter of configuration. This allows for additional processing by means of 3rd party tools which intercept messages, filter or adjust them, as well.

Results:

The time to implement and test the adapter was 16h in total. This includes unit testing and performing test #10, which tests the actual implementation performance of the

[136] An open source message queuing solution https://www.rabbitmq.com/

new adapter (see section 6.11 for details) and therefore provides vital data for the implementation's success.

Analysis:

As the research question is how easy and straightforward such a new adapter, to extend the DBGA, can be built no additional parameters were needed.

A general result of the test is that the gateway system can be easily extended. The test implementation – including all tests and test #10 - took around 16h to complete, which is very acceptable for a new protocol in such an environment.

6.11 Test #10: MS-MQ adapter performance to evaluate usage in business workflow environments

Research question:

As the new MS-MQ adapter, which was introduced in section 6.10, has to have a thorough evaluation with regards to performance and stability, to not compromise the entire DBGA, it was mandatory to know how these factors perform and scale under load.

Description:

In the test, a client writes sensor values to an actuator via the gateway using MS-MQ for all communications between client, server and actuator. The data to send to the actuator is placed as a XML- message in the server's in-queue where the server retrieves the message, processes it and a response is written to the server's out-queue, where it is read again by the client (so a kind of synchronous operation takes place with send – process – acknowledge).

The actuator is informed by placing a XML-message into the server's actuator out-queue where it can be read by any client.

For the test, the server was extended using a MS-MQ handling facility (plug-in) and the actuator notification was extended with another communication handler to use MS-MQ (aside HTTP REST). The queue names used are configurable and thus a scaling out to any infrastructure is simply a matter of configuration. This allows for additional processing by means of 3rd party tools which intercept messages, filter or adjust them, as well.

Test parameters:

To allow for enough data to remove any test setup influence 1,000 requests were performed for each test.

To remain as close to a realistic environment data producer and consumer were on one machine and the device gateway server on another.

Results:

To test the implementation several performance tests were undertaken. In Figure 6.14 the total time from posting the XML packet into the in-queue of the server until the response is returned to the client using the out-queue of the server is measured in msec.

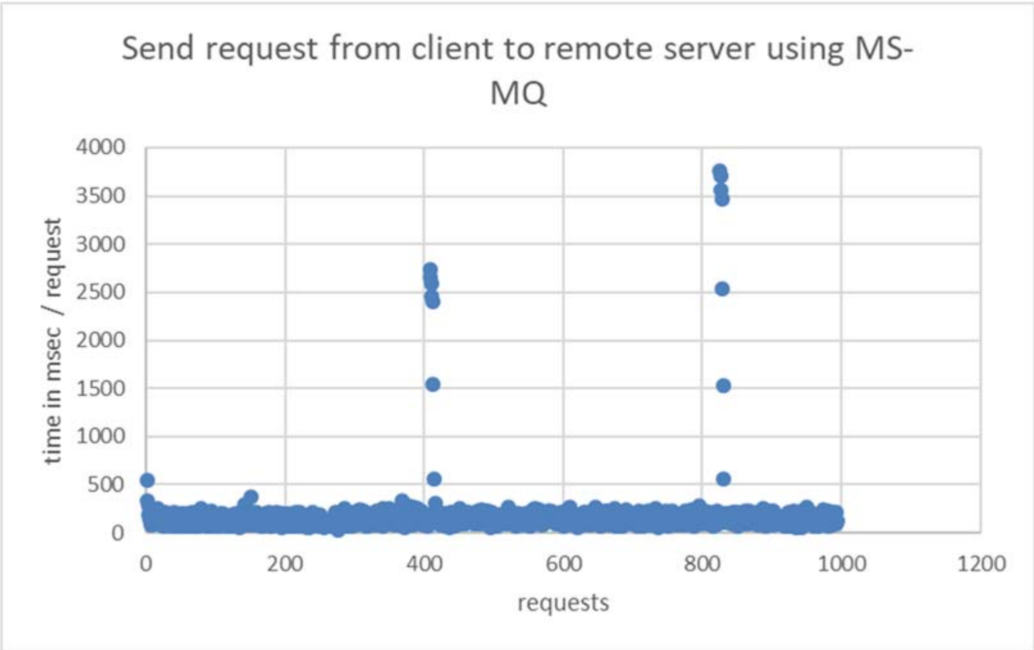

Figure 6.14: time in msec for sending a request including receiving response using MS-MQ from client to server

Figure 6.15 shows the time spent in the server (in msec) for processing inbound requests, forwarding another message to the actuator's out queue (which has to read the request) and passing back a message to the client. So here the initial client transmission of the originating data package is omitted and only the server processing time considered.

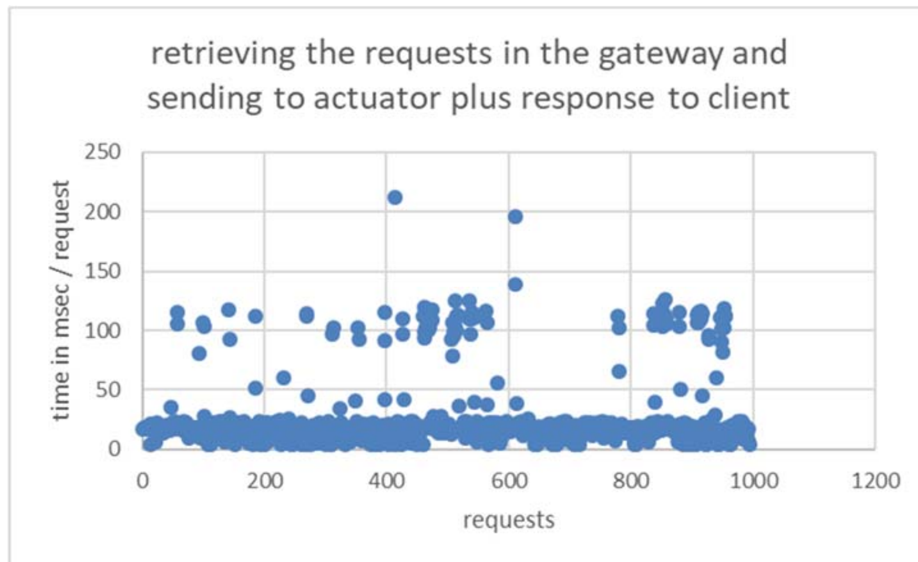

Figure 6.15: time in msec for receiving a request, processing and posting an actuator request and response packet using MS-MQ on the server

Analysis:

The usage of MS-MQ is quite straightforward and can be utilized for the gateway without problems. Special care must be taken (in a later state, not as part of this thesis) to not "overrun" an actuator with messages in case the actuator does not retrieve them from its queue. This could be done by using a TTL value (time to live) after which messages are simply discarded (optionally including messages after the failing message) by the messaging infrastructure.

The timing for MS-MQ is smooth and uniform over a wide range, with some exceptions which are not reproducible and occur in various places. A deeper investigation with a Memory and Performance analyzer shows that these are due to the load issues of the system, especially when the garbage collector is run which happens from time to time as many small objects are allocated and released frequently. Anyhow for asynchronous operations this does not matter very much as for each message sent the delivery is guaranteed, and therefore a simulation of a synchronous operation (send with acknowledge) is not necessary.

6.12 Overall observations during the experiments

After running all tests on the reference implementation, several general observations, not specific to a particular test, have been made:

- The architecture scales very nicely with regards to multi-threading and multi-processor issues (see section 6.4 and 6.7 for details) as regardless of the load

the system was still receptive (no stalling) and other processes could run as well, so system usage is very good.

- The same applies to system resource consumption as connections, memory and processing power are taken as needed and returned whenever possible (details especially in sections 6.4, 6.5 and 6.8). Not a single "out of memory" or "out of resource" situation has occurred[137].
- Handling of 250,000 internal sensor definitions with 100 historic data values for each sensor (therefore a total of 25,000,000 data items in the online cache at runtime) resulted in a system load of 1.4 GB of memory with a linear processing time curve (see sections 6.2, 6.3, 6.4 and 6.5). Therefore, adding more sensor definitions and / or historic values / sensor in the cache can be performed as needed where the maximum is just defined by the available memory of the system.
- The limits of the system were tested and currently the main limit would be networking (see sections 6.6, 6.7 and 6.9). As the transmission speed for local networks is usually fixed the only option to alleviate this bottleneck is to re-use and / or cache connections where the setup takes a long time or to use even more asynchronous patterns.
  Using such patterns has the added problem that producer and consumer, due to being disconnected from each other, require much more coordination and external control (by means of queue checks for pending requests, etc.) which complicates such systems again.

6.13 Discussion

The goal for this chapter was to conduct evaluation tests upon the reference implementation. These tests were designed to establish basic performance characteristics and check that requirements were fulfilled by the reference implementation.

In Table 6.12 the results are abbreviated for each test:

[137] Except in the ODATA sample where the XML parser (which can be substituted) returned an "Out of Memory"-exception after retrieving 500,000 records

Table 6.12: Test result overview

Test #	Short description	Results
1	Timing to push values into the core with dynamic calculations	With 200 sensors firing once every 10 msec the total data changes would be 20,000 / second
2	Timing to calculate virtual values in the core using only internal data (available measures)	Care must be taken when using triggers external to the DBGA. If 100 virtual sensors are queried every 100 msec resulting in 1,000 requests per second the system has a CPU load of around 1-2% (depending on the implementation).
3	Concurrent access (READ / WRITE) by several clients to check for concurrency, race conditions and locking issues	Parallel access from several execution threads with intermixed read and write works fine and is very stable. For larger scenarios SQL should not be used as an environment where triggers are implemented as the total execution time might simply be affected too much
4	Test with different data formats and conversion vs. native storage / handling	Differences among the different data formats and the conversion (the Dynamic variants) are not really existing which proves that storing the data in a non-native format is of no concern
5	Core timing considerations when using workflows (triggering, execution control for long-running tasks)	It is shown that agents are perfectly usable and capable to fulfil the needed functionality when considering the timing and load issues involved
6	Communication mode (REST, WCF, .NET Remoting (Binary)) implications	All communication protocols can be used for operations, yet some care must be taken with the overhead generated by SOAP messaging in heavy duty environments with a lot of traffic
7	ODATA access	Up to 8,000 result records the total time request time is still under 1s. For higher result sets the memory consumption of ODATA causes some concern and problems as for instance at 510,000 records an Out of Memory exception occurred. This has to be considered in real-life scenarios
8	Writing actuator values (by POSTing to a URL as a receiver)	Writing is possible without problems in all protocols. In UDP usage scenarios, special care has to be taken with packet sizes exceeding the MTU (very likely with verbose JSON data), as message segmentation is an issue.
9	MS-MQ adapter to prove the extendibility	The DBGA is easily extendable and the effort for doing so is within reason
10	MS-MQ adapter to READ / WRITE data – performance consideration	The usage of MS-MQ is quite straightforward and can be utilized for the gateway without problems. Mechanisms like TTL should be implemented so that a receiver, which is not taken packets from the queue, is not overrun by pending packets at the next start.

As all the tests upon the reference implementation were successful, the reference implementation holds promise from a performance perspective in that it has been

shown to scale according to the requirements set out and had no serious issues or problems.

In addition, due to (still) ever increasing speed, parallelism of the CPUs and advances in available memory these limits are getting higher and higher with each new generation of hardware. This means that more workflows and dynamically executable code can be put on a device gateway, making it even more dynamic and flexible.

Having established basic performance characteristics of the reference implementation of DBGA in this chapter, the next chapter discusses the implementation of the two diverse example scenarios outlined in section 3.1, and compares implementing these scenarios using the reference implementation of DBGA with that of implementing the scenarios with Microsoft Azure IoT.

7 Case Studies: comparing DBGA and Microsoft Azure IoT

In the following case studies, the 2 example scenarios defined in section 3.1 were implemented using the Device-Business-Gateway Architecture (DBGA)[138] reference implementation platform and using the Microsoft Azure IoT[139] platform. The scenario implementations were developed on both platforms by the author of this thesis.

They were then evaluated based on this implementation experience using the business process improvement criteria presented in section 3.7. The other state of the art approaches described in section 2.6 were also evaluated, but only based on the author's paper analysis of what would be involved with implementing the scenarios using those architectures.

The Microsoft platform was selected as being a message-based system, it would operate quite differently, and would be an interesting comparison.

Recall that a feature comparison with a number of the key gateway architectures was presented in section 2.6.1. While both implementations (DBGA and Azure IoT) share some similarities like the use of the simulation processes for data generation, the internal processing and overall architecture is quite different due to differences in message / event processing as well as data access options.

In general, with Azure IoT, like with most complex systems, many alternative implementations are possible. The chosen implementation was simply the most straightforward and efficient one to undertake. Whereas others might be superior in performance, throughput, cost or system utilization, the key focus of the evaluation was to evaluate particularly against the business process improvement criteria (as defined in section 3.7, and as highlighted as the goal for proposed architecture in our research question section 1.3) and to have a comparison for the DBGA with respect to those.

Section 7.1 presents how the Corrosion Lab Scenario was implemented using DBGA and Azure. Similarly, Section 7.2 discusses the implementations of the Exhibition Scenario. Section 7.3 then compares the implementations based on the criteria defined in section 3.7. In addition, this comparison also evaluates the other selected state of the art architectures with respect to in theory what would be involved in their implementation

---

[138] Available at: https://github.com/mglienecke/DeviceGateway

[139] Only results are available as access to the Azure IoT solution would incur costs

of the scenarios. Key findings are then presented in section 7.4, followed by a general conclusion in section 7.5.

7.1 Corrosion Lab Scenario

Using a subset of the environment described in section 2.5 an evaluation scenario using just two similar climate control chambers for salt brine tests[140] and the area before the chambers is defined. This is sufficient (it is argued) to show the problems and necessary work as the other chambers are somehow similar (using different protocols and different processes but otherwise more or less the same).

The actual numbers used in this scenario do not reflect real numbers yet were used to make the scenario more testable and usable. To have no data from a chamber for 10 seconds would be no problem in real life, whereas 15 minutes would be; yet to test for 15 minutes in a simulation has no impact on the outcome, therefore the shorter period of 10 seconds was taken.

As simulation setup the following assumptions were taken:

- There are 100 parts in the system;
- Each part to be tested is equipped with a RFID tag;
- There are 2 equivalent chambers in the system implemented as simulators which are queried similar to a chamber (using TCP/IP);
- The chamber simulators provide 3 independent sensor values (humidity in %, temperature in K, and salt brine concentration in % (as g/ 100 ml)).
  At random points in time erroneous data as well as no data is delivered to simulate sensor failure.
- RFID is provided as a simulator;
- For RFID the coordinate system is divided in 3 zones:
  - Zone A = Chamber A
  - Zone B = Chamber B
  - Zone C = Area before the chambers (preparation area)
- A part can be in 4 different zones: A, B, C or undefined;

---

[140] The chamber can be heated to +70° and cooled to -40° with a salt brine concentration from 5 - 40% simulating different parts of the world. Usually the chamber would be cooled down; salt brine applied, then heated, cooled down, etc. for 144 - 2880h in cycles. This simulates roughly the stress normally occurring during 3-4 years.

- For setup 30 parts are in Zone A, 20 in Zone B, 20 in Zone C and 20 undefined with 50 parts belonging to one test order (run in chamber A) and 50 to another test order (run in chamber B);
- The RFID simulator generates movement information for parts in a random way every second so that parts can move in and out of zones (and undefined state). These movements are sent to the gateway.

This scenario, being the more complex of the two scenarios, was quite straightforward to implement using DBGA, yet took a considerable amount of effort to implement in Azure as many different components had to be incorporated and adjusted for the overall solution.

In the following sub-sections, the specific details of implementation of the scenario using the DBGA reference implementation and Azure are discussed.

7.1.1 Device-Business-Gateway (DBGA) based implementation

The DBGA handles all part positions, Q-factors and chamber data as sensor data (virtual and real) with the corresponding device and sensor definitions. A part is a device with a sensor value for position and Q-factor; a chamber a device with sensor values for humidity, temperature and salt brine concentration. The alerting is implemented as an actuator where data can be written to (to generate the alert).

Chamber data values are provided by a simulation process which is scanned (pulled) periodically (every second) from the DBGA and returns arbitrary channel data (sometimes erroneous). This incoming data is then analysed by a WWF[141] code activity (chamber data validation), which is automatically run by the DBGA for every inbound new data item, and in case the temperature is invalid information is written to the alert actuator. As an alternative, the data validation could have been written as Python or C# script as well, yet the use of WWF code activity is very efficient and elegant as those encapsulate a unit of work very nicely and especially provide a good testing environment.

Part positions are generated by an RFID part movement simulation process (sometimes erroneous) for each part and send (pushed) to the DBGA. Their incoming data is

[141] WWF = Windows Workflow Foundation. Base classes and framework to create workflow applications in .NET environments

analysed by another WWF code activity (sensor position validation) and errors are written to the alert actuator.

The Q-factor for each part is calculated at request time (which is controlled by the definition of the virtual value, could be based on a change of a dependency sensor as well) using a WWF code activity as the Q-factor is registered as a virtual sensor with an underlying calculation.

In Figure 7.1 this DBGA-based implementation and the logical data flows are shown as a schematic overview with all participants and modules involved. Everything inside the DBGA is running on the server, both external processes on clients.

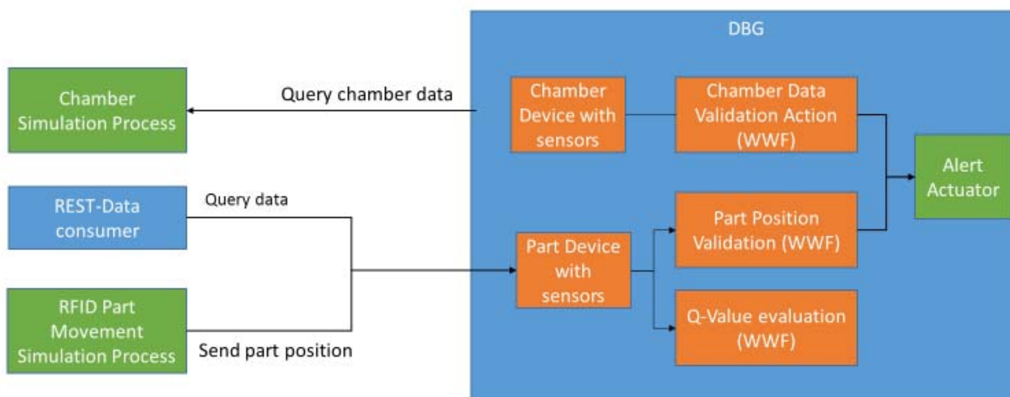

Figure 7.1: Schematic overview Corrosion Lab scenario using DBGA

7.1.2 Azure IoT based implementation

The IoT Hub is the main entry point for any message (event) into the system in Azure IoT, from which all further processing originates. Therefore, the chamber simulation process as well as the RFID part movement simulation process both push messages (some erroneous) into this (same) hub for further processing.

As Azure supports no pulling of data from devices (all data must be pushed) the scenario's requirement to scan the chambers is implemented insofar as the simulation process would scan a chamber and then push the readings further on to the hub.

A stream analytics job is receiving all inbound messages sent to the IoT hub and based on the sending device id (either "chamber_xxx" or "part_xxx") data is forwarded to a service bus with 2 event hubs for parts and chambers. An alternative implementation

could have been to send data directly to the destination event hubs, yet with the stream analytics in between additional filtering and data processing can happen, which would be beneficial in a real world usage.

These inbound events from the event hubs are then read and processed by a worker task running on an external system to Azure. Valid data is used to update an Azure SQL database with information needed to calculate the Q-factors, invalid data is written to an alerting service bus where it can be consumed. As an alternative implementation this worker process could have been implemented as an Azure process as well, yet this would require an additional virtual machine with the process running, configuration, etc. which would make it even more difficult to maintain and deploy. Therefore, the implementation running locally on the client was the most efficient for the time being.

As Azure IoT does not provide a facility to query a sensor value, this query mechanism had to be implemented using a REST-service. Therefore, the Q-factor calculation was implemented as a REST-API-service using MVC and is hosted inside Azure as a WebApi-service, which accesses the same Azure SQL database with the raw data for the calculation. Any client can then access the Q-factors as well as positions based on a simple REST-API.

In Figure 7.2 this Azure IoT-based implementation and the logical data flows are shown as a schematic overview with all participants and modules involved. Everything inside the Azure IoT box runs in the Azure cloud, other components on separate systems.

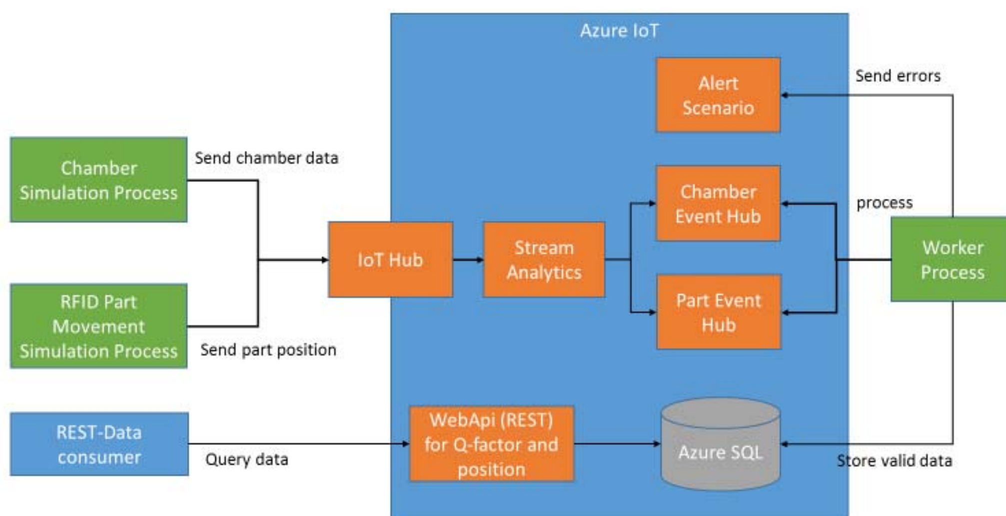

Figure 7.2: Schematic overview Corrosion Lab scenario using Azure IoT

7.2 Exhibition visitor congestion display system

In exhibitions people very often create clusters, whereas other areas are vacant (yet typically become clustered soon afterwards). So, the goal of this scenario is to show people, using their smartphones as devices, a live heat-map of the exhibition's space congestion and clustering. This allows the users to guide themselves either around hot-spots (clusters of congestion) or willingly proceed into such a hotspot.

To use such a very different scenario from a classical automation environment (like the one presented in section 7.1) is useful as it allows to test the DBGA in a totally different environment, which improves the usability in real-life environments, too. In addition, the problem to detect objects (here people) in an environment and especially their grouping and ways to avoid it are classical problems in industry automation as well – especially in areas like chaotic warehousing, storage of partially finished goods and provisioning of raw material to production lines.

For simulation setup the following assumptions were taken:

- The coordinate space for the scenario is defined as a 100 x 100 rectangle[142] ;
- 100 virtual people are simulated as this produces some load and reflects a medium sized exhibition event;
- They move in the coordinate space in arbitrary random directions with random speeds;
- To allow the simulation to operate smooth an iteration cycle is 1 second, so each second up to 100 virtual people move;
- Each second the position of the person is transferred from the person's device to the server;
- A simulation process is used instead of a real smartphone to send and receive data;
- The overall heat map is available as a virtual sensor value in JSON format with each data tuple defining X and Y coordinate of the point and the number of people.

[142] Any X/Y-dimension can be used, yet with 100x100 and 100 people the chance to have clustering rises

In the following sub-sections, the implementations for the Exhibition visitor scenario, using DBGA reference implementation and Azure IoT is described.

7.2.1 Device-Business-Gateway (DBGA) based implementation

In the DBGA implementation each person moving is implemented as a sensor which has a location value (X/Y) and the heat map as a single sensor value which returns a JSON object for the cells of the grid (10x10 raster) and the number of people in each cell.

The smartphone simulation process pushes the coordinates (using out-of-bounds values as invalid values as well) towards the DBGA where each value is validated using a WWF code action (position validation). In the DBGA, using a cyclic WWF action (heat map calculator) the value for the heat map is generated every second and can be retrieved using any method to query the sensor value for the heat map.

Using an UML sequence diagram this whole interaction between smartphone simulation process and DBGA with the server-side logic is shown in Figure 7.3:

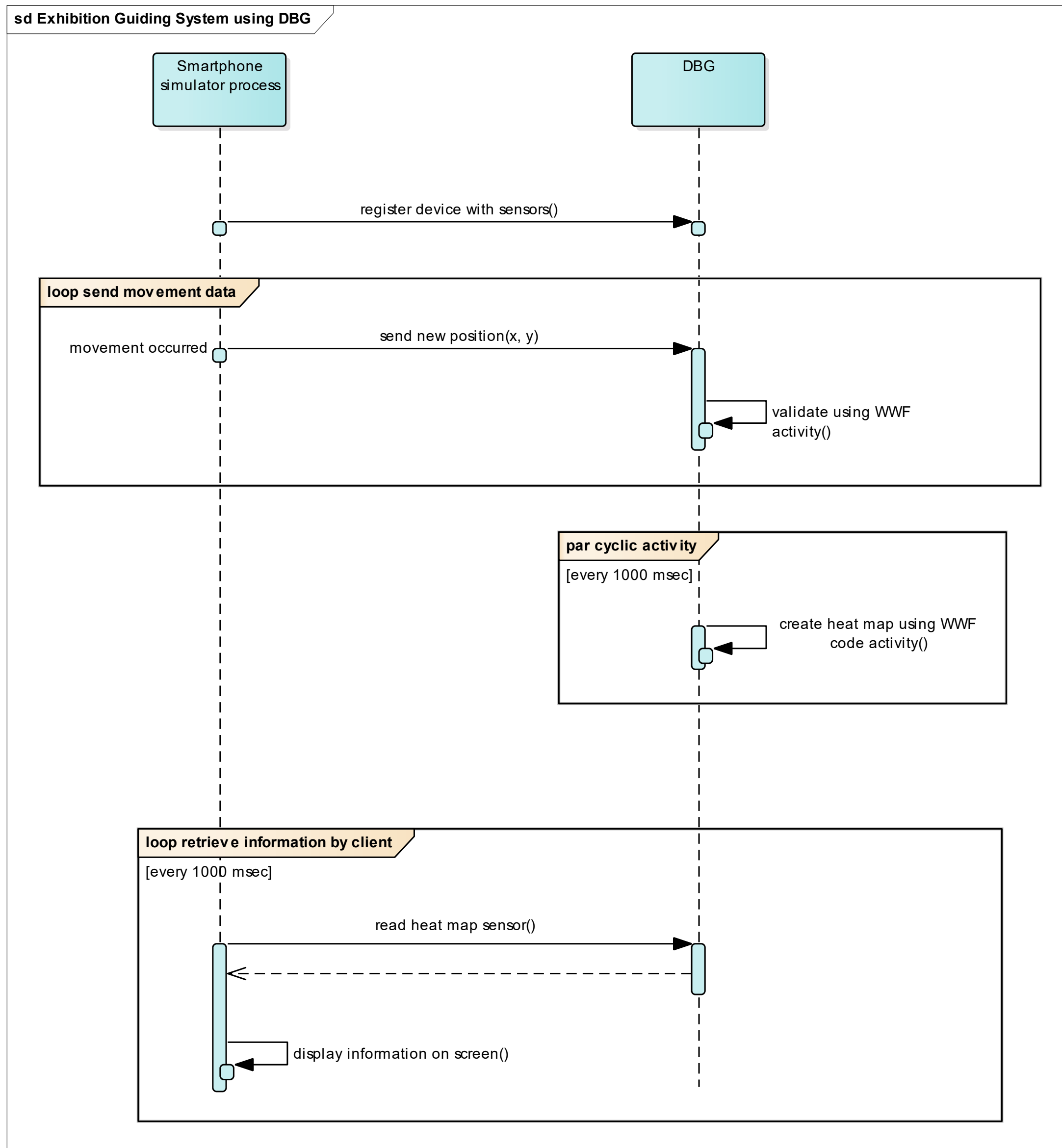

Figure 7.3: Exhibition Visitor Scenario implementation using DBGA

7.2.2 Azure IoT based implementation

In the Azure IoT implementation a smartphone simulation process pushes movement messages towards the Azure IoT hub. In the subsequent Stream Analytics module, which processes any inbound messages to the IoT Hub, valid movements are forwarded to an event hub "validsensordata" in a service bus, all invalid items are forwarded to the "invalidsensordata" event hub.

Using this approach, the stream analytics is capable of removing any unwanted messages directly so further processing is not needed. The valid messages are then consumed by an external (to the Azure IoT cloud platform) worker process, where the heat map is calculated and made available using a queue with a topic "newheatmap" which allows any interested subscriber to directly subscribe for new data. The other

benefit by using the subscriber mechanism is that messages in the queue are duplicated for each subscriber, so everybody receives the same message.

The alternative for the message queue approach would have been to provide a WebApi solution as in scenario 1, yet this seemed too much effort for such little benefit, as subscribing and reading from a queue is a straightforward process, if the client side API is present for the target device.

Using an UML sequence diagram this whole interaction between smartphone simulation process and Azure IoT with the cloud and worker process logic is shown in Figure 7.4:

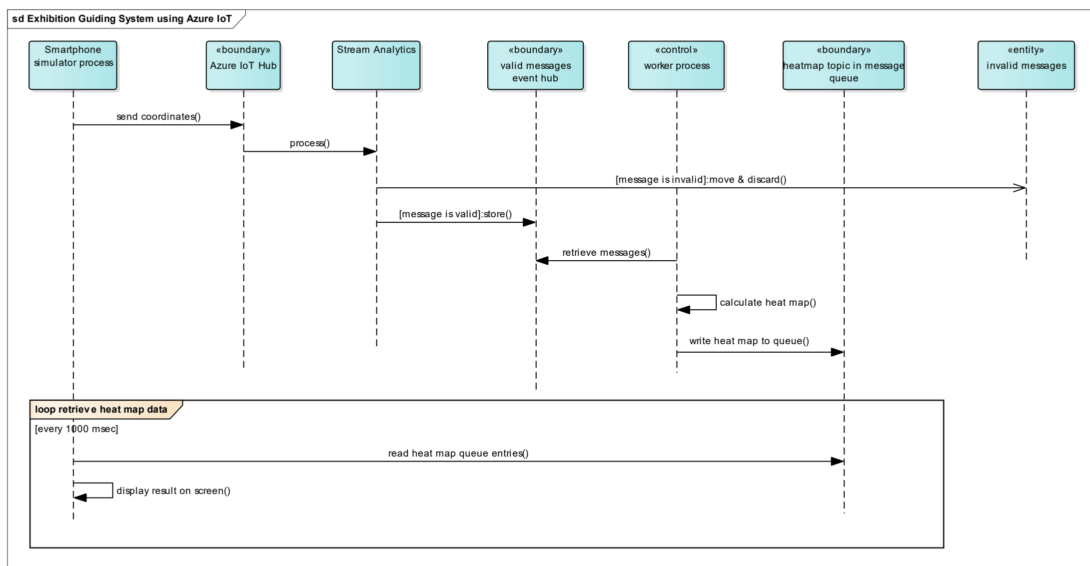

Figure 7.4: Exhibition Visitor Scenario implementation using Azure IoT

7.3 Comparison of architectures regarding the case studies

To evaluate the different possible architecture approaches, the defined evaluation criteria of Table 3.9 were applied to the selected state of the art architectures discussed in section 2.6. The resulting comparison table can be found in Table 7.1.

For DBGA and Azure columns, the evaluation is a result of the thesis author's experience gained in implementing the scenarios using the Azure and DBGA architectures. For the other architectures, the evaluation is purely based on a desk-based evaluation using the author's knowledge and experience to apply the criteria as if implementing the scenarios using the architectures.

Table 7.1: Comparison of different architectures

#	Criteria	DBGA	Azure IoT	Xively	sMAP	Custom implementation
M1	Level of achieving the requirements	3 - everything was achieved	3 - everything was achieved	2 – most requirements; some need a change in structure	1 - the second use case could not be really implemented and the 1st one not very well	3 - everything was achieved
M2	Flexibility offered by the architecture for an integrator to integrate devices into business processes	3 = highly flexible	2 = moderately flexible as the API is by design limited to several protocols which are not agnostic in their semantics. New protocols can be added, yet this is again custom implementation and a big task	1 = very inflexible Just the supported protocols with no possibility of change or added support for other protocols.	1 = very inflexible Only REST	2 = moderately flexible As it is custom made it will be a perfect match for the original environment, yet changes might require additional support
M3	Performance	3 – very fast	2 – Due to the external worker processes a lot of time is spent communicating with Azure IoT. This could be changed by placing everything in cloud modules, yet the simple connection	3 – very fast	3 - very fast	3 - very fast

#	Criteria	DBGA	Azure IoT	Xively	sMAP	Custom implementation
			initialization from device to Azure is quite slow.			
M4	Maximum number of devices / sensors / actuators supported	Depending on RAM and Hard disk in principle unlimited – in realistic environments around 100,000 sensors	In principle unlimited	In principle unlimited	Several 1000s of sensors	In principle unlimited
M5	Maximum number of sensors which can be handled as peak	Around 50,000 given a modern server	Depending on the CPU model used for the hosting nearly unlimited	Nearly unlimited	Several 1000s of sensors	In principle unlimited
M6	Complexity	1 – simple and straightforward	3 - highly complex	2 – moderately complex	2 - moderately	1 - easy, straightforward; very few connections and interactions
M7	Skill Required	2 - after the initial knowledge things get easy	3 - very high skill requirements	2 – moderate	1 - concepts are simpe and implementation is easy	3 - needs a lot of knowledge
M8	Learning Curve	2 - moderate	3 - very high learning curve	2 – moderate	1 - concepts are simply and implementation is easy	2 - moderate
M9	Time required building	1 – fast (all together 16 h for both scenarios)	3 – around 40h for both scenarios.	60 – 80h (change + external modules)	N/A	N/A
M10	Time required operating and administrating in one year	12h – mostly backups	80 h (quite complex tasks involved)	8h (all is hosted and maintained externally)	12h	8h

#	Criteria	DBGA	Azure IoT	Xively	sMAP	Custom implementation
M11	Ease of change in future	2 – moderate	3 – change is complex and requires a lot of effort, especially due to various cloud parts working together	2 – moderate	2 – moderate	3 - requires quite a lot
M12	Preservation of investment	2 - moderate	3 - very high preservation	3 – very high preservation	2 - moderate	1 - will cost more
M13	Free software / commercial software - cost	Free	Monthly costs are high	Monthly costs are moderate	Free	Onetime build cost
M14	Security	2 - can be made moderately secure	3 - can be made absolutely secure	3 - can be made absolutely secure	1 - there is no real concern about it	2 – moderate / depending on implementation
M15	Reliability / Fault tolerance	2 - can be made totally reliable	3 – is reliable	3 – is reliable	2 - not reliable except database	2 – moderate / depending on implementation
M16	Overall stability	2 = moderately stable Failing devices, sensors and actuators cause no problem. Failure of the database would make the system useless. Fixes are usually easy and can be directly done.	1 = highly stable Failing devices, sensors and actuators cause no problem (either no messages arrive or none are consumed). Everything is provided as highly available.	1 = highly stable Like Azure IoT	2 = moderately stable Failing devices and sensors are handled; no support for actuators. Failure in the database can cause a problem (like in the DBGA).	2 = moderately stable Like the DBGA

#	Criteria	DBGA	Azure IoT	Xively	sMAP	Custom implementation
			In case a real problem happens it is up to Microsoft to fix it		Fixes are usually easy and can be directly done.	
M17	Easiness to obtain the current device / sensor state and use it in the business process	3 = very easy Sensor state is kept directly and can be readily retrieved	1 = very hard There is no concept of "current state" of a sensor / device and it has to be constructed by the integrator	1 = very hard There is no concept of "current state" of a sensor / device and it has to be constructed by the integrator	Like the DBGA	Depending on the implementation, mostly like the DBGA
M18	Footprint	1 = low footprint	3 = very high footprint Not obvious, as all is "external", yet the footprint is very high due to various different services & applications needed	2 = high footprint Not obvious, as all is "external", yet the footprint is visible	1 = low footprint	1 = usually low footprint

7.4 Key Findings

After implementing the example scenarios using DBGA and Azure IoT, as well as comparing them to hypothetical implementations for the other selected state of the art approaches (Xively, sMAP and custom implementation) the following general observations have been derived (based on Table 7.1):

- sMAP has a specific usage pattern by concept and design which simply does not work very well with the scenarios presented. If the requirement would have been to, for example, to just display the chamber data over a period of time sMAP might have been a very powerful tool as exactly this is its core competence (including the querying part) - record time-series based values and retrieve them using various criteria. This non-match is even more obvious for the kind of transactional processing required by the scenarios
- Custom implementations are, like most tailor-made solutions for a problem, usually very efficient, fast and small in terms of system requirements as well as runtime overhead. At least this is true in the beginning, yet usually there is a general maintenance as well as extendibility and change problem over time as every change needs additional workforce (with associated labor cost). There is neither a new feature you get through an upgrade path, a bug fix, etc. Therefore, usually these custom solutions (good as they might be) have to be repeated (and re-done) every 4-6 years based on the author’s experience. In addition, a custom implementation usually is custom for one problem – if the problem changes (even if only slightly), the implementation does not fit any more (which might be the case with a more generalized tool). In addition, there is a tight coupling to the provider of the solution, which in case of for example company termination, could cause major problems, whereas in case of a general solution usually a replacement for the integrator can be found
- Xively is a nice moderately small and easy to use system with a straightforward design and architecture based on and around a messaging infrastructure. This simplicity is a wonderful thing if it suits the needs of a device integration, yet in case of the usage scenarios it would not match and a lot of extra work would be needed to get the case study implemented. In addition, as there are no extra modules or services such as those provided by Azure IoT (which is quite similar in the base messaging engine) the usable tool-set is quite small.
  As it is a hosted solution in the cloud any user has to have no concerns regarding data backup, stability and availability as these can be considered assumed.

- Azure IoT can be used to implement most real usage requirements, yet quite often the requirement must be adjusted to the specific implementation patterns offered by Azure, especially the message based processing. Being a large and sophisticated platform, especially when additional services like storage, real-time analysis and message processing are involved, makes it a very complex, hard to learn and maintain solution. It might offer all which is needed, yet to get the solution right requires extensive knowledge very few people will have. In addition, due to its mainly web-based user-interface for administration (when avoiding the scripting interface) many tasks are not that easy to accomplish as many features are hard to find or not present in their entireness (like for example the event hubs for service buses which are only available in the old, not the new version of the management portal). In addition, like Xively, being a hosted solution, availability, data backup, etc. can be taken for granted. Yet the cost involved is quite essential as there is a fee for transactions or processing time which adds up to considerable amounts over time.
- All message-based systems under investigation (Xively, Azure IoT) as well as others (like AWS IoT (amazon) or Google IoT) only process messages and have no notion of a device or a sensor as such. For them these are just message start- or endpoints with some centralized (workflow) processing in between, yet no notion of a sensor state exists, which means every integrator, who needs to have a current state of a sensor has to build logic to provide this state information. This could be achieved by monitoring the state by scanning the messages and updating a database with some kind of web access, etc. Yet always it requires the integrator to do something extra.
- The DBGA offers all that is needed to implement the example usage scenarios with a straightforward and quite easy implementation approach. As the whole design is centered around the storage, access and calculation of sensor values, an integration can be done far more "naturally" from an integrator's point of view, as many tasks (like necessary in for example Azure) are either done automatically or are much easier to implement. In addition, due to the very extendible architecture and the many provided adjustment points any change and adaptation can be made quickly and easily, as well as new features added quickly.

As the source code of the reference implementation is available[143] there is a chance for every research project to just use the source and investigate or change as necessary.
Being based on software which is free of charge (including SQL Server Express[144], except Microsoft Windows, if not run under LINUX or MacOS) there are no additional costs to be considered and the hosting can be done from anything like a simple PC to a fully clustered environment.
Due to the varied communication protocols supported as well as the achievable internal knowledge gain by means of virtual values as well as workflows a business process integration is greatly enhanced as less work has to be done on the process side and more can be shifted to the gateway.

7.5 Discussion

In this chapter the details of actual implementation of the two scenarios from section 3.1 using the Device Business Gateway (DBGA) and Azure IoT were described and compared using important criteria for business process integration improvement defined in Table 3.9.

The result of the overall comparison presented included the other selected state of the art architectures (but where scoring was based on a hypothetical implementation approach) in Table 7.1 and then key findings discussed (section 7.4).

In short the main findings were that sMAP would not be usable for the example scenarios, Xively with some changes in requirements or re-structuring of the solution and custom implementations could do very well, yet there are other problems.

Azure IoT could very well handle all the requirements of the scenario, yet due to its complexity, different services involved and thus the various skill-sets required to build an integration solution, is probably more suited for larger or very large environments. Especially, given the specific nature of Azure IoT with a messaging system in the core produces a different approach to most problems, which very often is not suited exactly to typical environments. This is very obvious when it comes to maintaining sensor states as a basis for further action.

---

[143] https://github.com/mglienecke/DeviceGateway

[144] Which has some limits, but these are not likely to come into effect

It has been found during the evaluation of the implementation of the example scenarios that the DBGA can satisfy the requirements nicely and with very little effort, which in addition to its lean and extensible architecture makes it very usable.

A general overall observation from the implementation of the scenarios is that DBGA suits much better towards an industry automation type of project than would Azure IoT or Xively.

Having evaluated the different approaches towards device gateway integration into business processes and made comparisons in this chapter, the next chapter will present overall conclusions.

8 Conclusions

The research question posed in this thesis is to what extent a new device gateway architecture (with integrated data quality control, increase of data value by semantic enrichment, communication interface agnosticism, data target independence and data format agnosticism) that is available for low cost, will improve business process integration.

In this thesis, the Device-Business-Gateway Architecture (DBGA) has been proposed as that new architecture, motivated by an analysis of the state of the art. A design of the DBGA has been outlined in the thesis, and a reference implementation of that design has been described and made available for other researchers/projects to use. Evaluation of the proposed DBGA has been undertaken by using test cases to test the reference implementation, and case study implementations to evaluate the potential improvement in the business integration experience. To further test the reference implementation and its usability, as well as stability and performance, parts of DBGA reference implementation are already being utilized in the author's customer industry automation environment.

In the following sub-sections, a recap of the structure and flow of this thesis is presented (section 8.1), the main findings highlighted (section 8.2), future work postulated (section 8.3) and the contribution made by this research is summarized (section 8.4).

8.1 Structure of the thesis

To be able to properly research the question this thesis used the following structured approach.

Chapter 2 provided background information about which kind of devices, sensors, actuators and consumers are relevant (section 2.1) as well as industry automation (section 2.2) and Industry 4.0 (section 2.3). This was followed by the role of device gateways (section 2.4), a real-world business process integration example (section2.5), an overview of exemplary gateway architectures as state of the art (section 2.6) and a discussion (section 2.7).

Chapter 3 defined two usage scenarios, based on real-world requirements (section 3.1), which were later used to implement case studies (chapter 7). From these the standard use cases that any gateway architecture must be able to support (section 3.2) as well as the requirements (section 3.3 and section 3.4) and based on those the characteristic of the Device-Business-Gateway (DBGA) (section 3.5) were derived. A comparison of

requirement achievement of different architectures (section 3.6) was concluded by the definition which measurements are usable to define the effectiveness of improvement of an architecture (section 3.7).

As the use cases, requirements and characteristics were defined, chapter 4 detailed the actual design of the architecture, first by presenting data sources and destinations (section 4.1), followed by the supported communication options (section 4.2) and the usage of workflows in the design (section 4.3). Afterwards the components of the architecture in a logical view (section 4.4) were presented, followed by the process or dynamic view (section 4.5). The chapter finished by presenting the physical view (section 4.6) and data model (section 4.7).

Based on the definitions in chapter 4 the actual reference implementation of the DBGA was specified in chapter 5. Here the available options (section 5.1), the general structure (section 5.2) and then the individual components like the central service launcher (section 5.3), central server service (section 5.4), global data contracts between devices and server (section 5.5) and the base code for modules running on specific devices (section 5.6) among others were explained.

Using the reference implementation chapter 6 described nine tests (section 6.2 to 6.10) which were constructed to test several low level criteria of the reference implementation. Among these were the ability to handle how many requests / second, times consumed to validate data, evaluation of virtual values, concurrent access situations, race conditions, etc. In addition, several characteristics like data actuator support, data format agnostic, etc. were tested alongside.

Based on the example usage scenarios (section 3.1) chapter 7 described the actual implementation of these with the DBGA (section 7.1.1 and 7.2.1) as well as Microsoft Azure IoT (section 7.1.2 and 7.2.2). This was followed by a comparison of all architectures regarding implementation of the usage scenarios (section 7.3)[145] and resulting findings were provided (section 7.4).

Additional information related to, but not directly relevant for the mentioned topics, was made available in appendices (appendix A, B, C and D).

---

[145] here for the not implemented ones a hypothetical implementation based on desk analysis was used as a basis

8.2 Main Findings

Taking account of the findings of the case studies (section 7.4) and characteristics of DBGA developed (section 3.5) based on general requirements (section 3.3 and section 3.4) and use cases (section 3.2), the following main findings can be derived as follows:

1) Indications are that the DBGA is capable of improving business process integration in a variety of industry automation environments.

   Due to its design "around" the sensor with all necessary operations like state preservation, access, virtual value calculation provided, any integrator can focus on the process, instead of on the technical part of the gateway.

   This is achieved mainly by ensuring the architecture possesses the characteristics of: integrated data quality control, increase of data value by semantic enrichment, communication interface agnosticism, data target independence and data format agnosticism.

   It should be noted though that the DBGA does not claim to be a "Soft-PLC" or running in a non-stop (24x7) environment which would require much more dedicated tests, runtime analysis and further strengthening of especially the failover capabilities.

2) These architecture characteristics directly underpin the following indicators (see Table 3.9 or details), which enable the improvement in business process integration:
   a. Flexibility in the manner in which new devices are integrated into the process and preservation of sensor state
   b. Ease of change in the future
   c. Cost of change
   d. Operational cost per transaction

In addition, due to the simplicity and straightforwardness of implementation using DBGA, a business process integration can be modeled in exactly the way it is needed, not in the way which it may be constrained by adopting a commercial solution.

3) Message based solutions are very useful for (very) large scale device integration platforms where the (or a main) focus is on data investigation, collection and analysis (big data). This is usually the case for large-scale, geographically disperse projects, usually involving several distinct entities, like supply-chain-management and so on.
   Due to the internal structure these solutions are more concerned about how to route messages than about device or sensor specific tasks.
   Feedback to actuators or sensors in terms of centralized control is not easily implementable using such platforms and requires much effort

   The DBGA is no replacement for message-based solutions, yet can be used effectively as a gateway, which provides pre-aggregated and value-enriched data, for the message-bus. Another use-case for the DBGA would be as a consumer to scan for device messages on the message bus and aggregate them like normal sensor data – providing aggregated value further onto the value-chain.

-

4) Commercial solutions (Azure IoT, Xively, etc.) usually have a historic background in terms of a product they are based on, which clearly dictates their use and benefit, which often is not in line with the requirements of the device and sensor integrators.
   In contrast, the DBGA was especially designed to match these requirements and therefore – given the environment where it is beneficial – it is argued it will provide superior benefit.

5) Due to be being freely available the DBGA can be a significant contribution to research projects or classic business process integration in the industry automation environment as everything can be investigated, changed or adapted free of charge.

As many requirements are similar between IoT and automation environment, the DBGA could be applied to IoT scenarios as well. Yet care has to be taken as issues like power consumption, edge or fog computing, unreliable communication and other issues usually associated with IoT are considered only on a very limited scale or not at all.

8.3 Future Work

There are a number of areas where future work might be considered based on a variety of reasons which is briefly shown in Table 8.1:

Table 8.1: Future work overview

Area of future work	Reason
JavaScript -> Chakra integration	As JavaScript is gaining more and more momentum even in server-side processing the integration of a JavaScript-engine into the gateway (similar to the Python one) would be very beneficial for the ease of integration
Standard functions for virtual values	As very virtual values will be implementing functionality to calculate KPIs (key process indicators), it would be good to have an ability to define standard functions which can be used in the evaluation. Examples would be MEAN() or MIN(), MAX(), etc.
Default checking for Write (Upper / Lower Bound, Frequency, etc.)	In real projects the data cleansing is very often a repetitive task which could be greatly simplified if pre-definable filters (upper / lower check, bandwidth, frequency, patterns, etc.) could be simply applied.
Wider Ontology support	Extend the very basic ontology service layer so that it covers many more attributes and provides better information retrieval mechanisms.
Query & Discovery support	When applied to a wider audience, the previous points become relevant in terms of for example discovery which actuator might be able to alert something or which sensor could deliver the temperature in room XYZ
Access control	Especially for large scale environments using many contributors and / or sensitive data (for example medicine) further advances in access control would be necessary as it must be made sure that only authorized people access their data
Stream processing	All data from sensors so far was based on singular data points. As streams as data source are becoming more and more present (for example video data, music, etc.) a facility to process, handle and use in business workflows would be very

	interesting. Especially features like for example object recognition from video sources might prove extremely useful as then for example CCTV cameras could be used as data sources and the resulting value used to trigger events
Providing a miniaturized version for usage on a for example smart device	With the advent of smart devices like smart phones, Raspberry Pi, etc. it might be very interesting to have a miniaturized version of the proposed gateway on such a device so that the device could be used as the first step of integration in terms of decentral data cleansing, pre-processing, etc. The communication with a back-end system might then happen based on TCP/IP alone with local data storage only in case of link failure
Big Data integration	Provide facilities so that further data processing facilities like for instance data lakes can be automatically populated by the data generated from the device gateway (especially the virtual sensor values).
Further evaluate the research	Take the existing implementation and adapt it to different environments using various sensors and / or backend systems
Extend the concepts	Take the work as a basis for further conceptual work and research on behalf of business process integration, standard workflows and especially more time-constrained environments.
Compare the work with other implementations	Compare new and upcoming versions of other implementations (like IoT gateways, message bus architectures, cloud / fog approaches) to the work and adapt where needed and required.

8.4 Contribution

As this thesis provides a clear, simple and straightforward design for a device business gateway architecture (DBGA), alongside the background information and discussion contained, it is a good starting point for further research into more specific areas of the

business process integration of devices and sensors in industry automation environments.

Especially improvements in business process integration using autonomous operations or semantic value increase will benefit substantially from the research undertaken as the demand for sure will increase based on the current tendencies ([56], [84], [87]). This has been the focus of the author's publication [83], derived from this research, as well.

Using the provided reference implementation any interested integrator or researcher has access to a free of charge, fully functional device gateway which can be used as is or extended to cater for additional demands. With this ability the time and effort needed to integrate devices and sensors into research projects is greatly reduced and thus more effort can be put into added or new functionality and value-added services which will contribute to the research community as well.

Using the defined characteristic for the DBGA, derived from the requirements, which were derived from the use cases as well as the state of the art, is an additional contribution as these characteristic and requirements could be used by other researches in similar environments as well.

To allow for a wider impact and therefore contribution of the DBGA, it is planned, after the initial implementation (based on the DBGA) of a full-scale integration at the author's customer, to publish the findings, future improvements and benefits in various publications.

To further increase the visibility of the DBGA, participation with papers and / or poster sessions at SenSys 2019 (http://sensys.acm.org/) CASE 2019 (http://case2018.org/) or ICRA 2019 (https://www.icra2019.org/) are planned.

9 References

[1] Aberer, K., Hauswirth, M., & Salehi, A. (2006, September). A middleware for fast and flexible sensor network deployment. In Proceedings of the 32nd international conference on Very large data bases (pp. 1199-1202). VLDB Endowment.

[2] Schramm, P., Naroska, E., Resch, P., Platte, J., Linde, H., Stromberg, G., & Sturm, T. (2004). A service gateway for networked sensor systems. IEEE Pervasive computing, 3(1), 66-74.

[3] Pautasso, C., Zimmermann, O., & Leymann, F. (2008, April). Restful web services vs. big'web services: making the right architectural decision. InProceedings of the 17th international conference on World Wide Web (pp. 805-814). ACM.

[4] Guinard, D., & Trifa, V. (2009, April). Towards the web of things: Web mashups for embedded devices. In Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain (p. 15).

[5] Stirbu, V. (2008, August). Towards a restful plug and play experience in the web of things. In Semantic computing, 2008 IEEE international conference on (pp. 512-517). IEEE.

[6] Pachube - data infrastructure for the Internet of Things. Available at: http://www.pachube.com/ [Accessed March 21, 2011].

[7] Nath, S. SenseWeb: An Infrastructure for Shared Sensing. In US-Korea Conference on Science, Technology, and Entrepreneurship (UKC).

[8] Open Data Protocol (OData). Available at: http://www.odata.org/ [Accessed March 21, 2011].

[9] Vasters, C., 2012. Internet of Things - Using Windows Azure Service Bus for ... Things! [WWW Document]. Using Windows Azure Service Bus for ... Things! URL http://msdn.microsoft.com/en-us/magazine/jj133819.aspx

[10] Internet of Things - A Smart Thermostat on the Service Bus [WWW Document], 2012. URL http://msdn.microsoft.com/en-us/magazine/jj190807.aspx

[11] SENSEI - Home [WWW Document], 2012. URL http://www.sensei-project.eu/ [Accessed July 29, 2012]

[12] HEU, A. B., HEU, P. G., CEA, A. O., & Stefa, J. (2013). Internet of Things Architecture.

[13] Trifa, V., Wieland, S., Guinard, D., Bohnert, T.M., 2009. Design and implementation of a gateway for web-based interaction and management of embedded devices. Submitted to DCOSS, 1-14.

[14] Haller, S., & Magerkurth, C. (2011, March). The real-time enterprise: Iot-enabled business processes. In IETF IAB Workshop on Interconnecting Smart Objects with the Internet.

[15] Prehofer, C., van Gurp, J., & di Flora, C. (2007, September). Towards the web as a platform for ubiquitous applications in smart spaces. In Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI), at Ubicomp (Vol. 2007).

[16] Debaty, P., Caswell, D., 2001. Uniform web presence architecture for people, places, and things. Personal Communications, IEEE 8, 46–51.

[17] Extended Environments Markup Language: EEML [WWW Document], 2010. URL http://www.eeml.org/

[18] Dawson-Haggerty, S., Krioukov, A., & Culler, D. E. (2012). Experiences integrating building data with smap. University of California, Berkeley, Tech. Rep.

[19] Aslam, M. S., O'Regan, E., Rea, S., & Pesch, D. (2009, June). Open framework middleware: an experimental middleware design concept for wireless sensor networks. In Proceedings of the 6th international workshop on Managing ubiquitous communications and services (pp. 35-42). ACM.

[20] Debaty, P., Caswell, D., 2001. Uniform web presence architecture for people, places, and things. Personal Communications, IEEE 8, 46–51.
[21] Vazquez, J.I., De Ipina, D.L., Sedano, I., 2007. Soam: A web-powered architecture for designing and deploying pervasive semantic devices. International Journal of Web Information Systems 2, 212–224.
[22] Priyantha, N.B., Kansal, A., Goraczko, M., Zhao, F., 2008. Tiny web services: design and implementation of interoperable and evolvable sensor networks, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems. pp. 253–266.
[23] Kakanakov, N., Stankov, I., Shopov, M., & Spasov, G. (2006). Controller Network Data Extracting Protocol–design and implementation. In Proc. CompSysTech (Vol. 6).
[24] Spasov, G., Stankov, I., & Petrova, G. (2006). WIRELESS REAL-TIME GATEWAY (WRTG) FOR EMBEDDED DEVICES.
[25] "Internet of Things - Architecture — IOT-A: Internet of Things Architecture". [Accessed 29. Juli 2012]. http://www.iot-a.eu/public.
[26] "EPCglobal | GS1 - The global language of business". [Accessed December 29 2013], http://www.gs1.org/epcglobal.
[27] "Open Geospatial Consortium | OGC(R)". Accessed December 29, http://www.opengeospatial.org/.
[28] "Public — EPoSS." Accessed December 29, 2013. http://www.smart-systems-integration.org/public.
[29] "EPoSS Strategic Research Agenda 2009 — EPoSS." Accessed December 29, 2013. http://www.smart-systems-integration.org/public/documents/publications/EPoSS%20Strategic%20Research%20Agenda%202009.pdf/view.
[30] "ChipworkX_Development_System_Broch_Pinout.pdf." Accessed December 30, 2013. http://www.ghielectronics.com/downloads/man/ChipworkX_Development_System_Broch_Pinout.pdf.
[31] "Tahoe-II." Accessed December 30, 2013. http://devicesolutions.net/support/legacyproducts/tahoeii.aspx.
[32] "Arduino - HomePage." Accessed December 30, 2013. http://arduino.cc/.
[33] "Netduino Home." Accessed December 30, 2013. http://netduino.com/.
[34] "AMI > Home." Accessed December 30, 2013. http://www.aug-electronics.com/ami.
[35] "SunSPOTWorld - Home -." Accessed December 31, 2013. http://www.sunspotworld.com/.
[36] Polastre, J., Szewczyk, R., & Culler, D. (2005, April). Telos: enabling ultra-low power wireless research. In IPSN 2005. Fourth International Symposium on Information Processing in Sensor Networks, 2005. (pp. 364-369). IEEE.
[37] Alliance, Z. (2006). Zigbee specification. .
[38] Siernens, A. G., Automatisierungstechnik, G., & Basis, I. S. O. 6.8 SINEC.Praxis der Automatisierungstechnik, 238.
[39] Semiconductor, P. (1998). The I^2C-Bus Specification: Version 2.
[40] Büch, C. (2006, June). SPI–Serial Peripheral Interface. In PhysiN-Seminar Universität Koblenz-Landau (Vol. 27). .
[41] "Standard, O. A. S. I. S. (2009). Devices Profile for Web Services Version 1.1.
[42] Zeeb, E., Bobek, A., Bohn, H., Prueter, S., Pohl, A., Krumm, H., ... & Timmermann, D. (2007). WS4D: SOA-Toolkits making embedded systems ready for Web Services. Open Source Software and Productlines 2007 (OSSPL07).
[43] Clarke, G. R., Reynders, D., & Wright, E. (2004). Practical modern SCADA protocols: DNP3, 60870.5 and related systems. Newnes.
[44] "Documents — IOT-A: Internet of Things Architecture." Accessed January 1, 2014. http://www.iot-a.eu/public/public-documents/documents-1.

[45] "OData Version 4.0 Part 1: Protocol." Accessed January 1, 2014. http://docs.oasis-open.org/odata/odata/v4.0/cs01/part1-protocol/odata-v4.0-cs01-part1-protocol.html.
[46] "Open Data Protocol (OData)." Accessed March 21, 2011. http://www.odata.org/.
[47] Kakanakov, N., Stankov, I., Shopov, M., & Spasov, G. (2006). Controller Network Data Extracting Protocol–design and implementation. In Proc. CompSysTech (Vol. 6).
[48] Uckelmann, D., M. Harrison, and F. Michahelles. Architecting the Internet of Things. Springer, 2011.
[49] Duquennoy, S., Grimaud, G., & Vandewalle, J. J. (2009, May). The Web of Things: interconnecting devices with high usability and performance. InEmbedded Software and Systems, 2009. ICESS'09. International Conference on (pp. 323-330). IEEE.
[50] Trifa, V. M. (2011). Building blocks for a participatory Web of things (Doctoral dissertation, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 19890, 2011).
[51] "Push API." Accessed January 6, 2014. http://www.w3.org/TR/push-api/.
[52] Modbus, I. D. A. (2004). Modbus application protocol specification v1. 1a. North Grafton, Massachusetts (www. modbus. org/specs. php).
[53] Meyer, S., Ruppen, A., Magerkurth, C., 2013. Internet of things-aware process modeling: integrating IoT devices as business process resources, in: Advanced Information Systems Engineering. Springer, pp. 84–98.
[54] Weske, M., 2012. Business Process Management: Concepts, Languages, Architectures. Springer Science & Business Media.
[55] Dawson-Haggerty, S., Jiang, X., Tolle, G., Ortiz, J., Culler, D., 2010. sMAP: a simple measurement and actuation profile for physical information, in: Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems. ACM, pp. 197–210.
[56] Internet of Things Market Size- Postscapes [WWW Document], n.d. URL http://postscapes.com/internet-of-things-market-size (accessed 8.8.15).
[57] Zachariah, T., Klugman, N., Campbell, B., Adkins, J., Jackson, N., & Dutta, P. (2015, February). The internet of things has a gateway problem. In Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications (pp. 27-32). ACM.
[58] Waher, P. (2015). Internet of Things-Sensor Data.
[59] Funes, P. (2001). Evolution of complexity in real-world domains (Doctoral dissertation, Brandeis University). .
[60] Manyika, J., Chui, M., Bughin, J., Dobbs, R., Bisson, P., & Marrs, A. (2013). Disruptive technologies: Advances that will transform life, business, and the global economy (Vol. 12). San Francisco, CA: McKinsey Global Institute.
[61] Worlds, G. S. P. the Internet of Everything Are Colliding to Create New Markets, November 2013.
[62] Thamhain, H.J., 2015. Management of Technology: Managing Effectively in Technology-Intensive Organizations. John Wiley & Sons.
[63] Kruchten, P., 1995. Architectural Blueprints—The "4+ 1" View Model of Software Architecture. Tutorial Proceedings of Tri-Ada 95, 540–555.
[64] Bass, Len; Paul Clements; Rick Kazman (2012). Software Architecture In Practice, Third Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[65] Erl, T. (2005). Service-oriented architecture: concepts, technology, and design. Pearson Education India.
[66] Robert Daigneau. 2011. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and Restful Web Services (1 ed.). Addison-Wesley Professional
[67] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
[68] Anti Pattern [WWW Document], n.d. URL http://c2.com/cgi/wiki?AntiPattern (accessed 12.5.15).

[69] Fowler, M. (2004). Inversion of control containers and the dependency injection pattern.
[70] Merson, P.F., 2009. Data model as an architectural view.
[71] Botts, M., & Robin, A. (2007). OpenGIS sensor model language (SensorML) implementation specification. OpenGIS Implementation Specification OGC,7(000).
[72] Franklin, S., Graesser, A., 1997. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, in: Intelligent Agents III Agent Theories, Architectures, and Languages. Springer, pp. 21–35.
[73] Sector, I.T.S., 1996. ITU-T Recommendation Z. 120. Message Sequence Charts (MSC96).
[74] Manyika, J., Chui, M., Bisson, P., Woetzel, J., Dobbs, R., Bughin, J., & Aharon, D. (2015). Unlocking the Potential of the Internet of Things. McKinsey Global Institute http://goo. gl/qzq5mV.
[75] Azure IoT Suite [WWW Document], n.d. URL https://azure.microsoft.com/en-us/solutions/iot-suite/ (accessed 5.22.16).
[76] What is Xively - Xively [WWW Document], n.d. URL https://xively.com/whats_xively/ (accessed 5.22.16).
[77] Leveraging the Internet of Things for Competitive Advantage. Knowledge@Wharton (2016, March 22). Retrieved from http://knowledge.wharton.upenn.edu/article/leveraging-the-internet-of-things-for-competitive-advantage/.
[78] W. Bolton, Chapter 1 - Programmable Logic Controllers, In Programmable Logic Controllers (Fifth Edition), Newnes, Boston, 2009, Pages 1-19, ISBN 9781856177511, http://dx.doi.org/10.1016/B978-1-85617-751-1.00001-X.
[79] TinyDB: A Declarative Database for Sensor Networks [WWW Document], n.d. URL http://telegraph.cs.berkeley.edu/tinydb/ (accessed 5.23.16).
[80] IrisNet [WWW Document], n.d. URL http://research.microsoft.com/pubs/76117/pervasive-03.pdf (accessed 5.23.16).
[81] Microsoft_Azure_IoT_Reference_Architecture.pdf [WWW Document], n.d. URL http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf (accessed 5.25.16).
[82] "Service Assisted Communication" for Connected Devices | Clemens Vasters. [WWW Document], n.d. URL https://blogs.msdn.microsoft.com/clemensv/2014/02/09/service-assisted-communication-for-connected-devices/ (accessed 5.26.16).
[83] Michael Glienecke, David Lewis, Declan O' Sullivan, A Value Added Device Gateway Architecture for Sensors and Actuators, *International Journal of Sensor and Related Networks (IJSRN)*, 1, (1), 2013, p31 – 36
[84] Vogel-Heuser, Birgit, et al. Handbuch Industrie 4.0 Bd.2, 2nd Ed: Automatisierung (VDI Springer Reference). Springer Vieweg, 2017
[85] "OPC-UA (Unified Architecture)" [WWW Document]. OPC Foundation (blog). accessed 3rd June 2018. https://opcfoundation.org/about/opc-technologies/opc-ua/
[86] "OPC-Classic" [WWW Document]. OPC Foundation (blog). accessed 3rd June 2018. https://opcfoundation.org/about/opc-technologies/opc-classic/
[87] Abele, Eberhard, und Gunther Reinhart. Zukunft der Produktion. Hanser München, 2011.
[88] Jammes, François, und Harm Smit. „Service-oriented paradigms in industrial automation". IEEE Transactions on industrial informatics 1, Nr. 1 (2005): 62–70.
[89] H. Bohn, A. Bobek, und F. Golatowski. „SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains". In International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning

Technologies (ICNICONSMCL'06), 43–43, 2006. https://doi.org/10.1109/ICNICONSMCL.2006.196.
[90] A. Cannata, M. Gerosa, und M. Taisch. „SOCRADES: A framework for developing intelligent systems in manufacturing". In 2008 IEEE International Conference on Industrial Engineering and Engineering Management, 1904–8, 2008. https://doi.org/10.1109/IEEM.2008.4738203.
[91] Razzaque, M. A., M. Milojevic-Jevric, A. Palade, und S. Clarke. „Middleware for Internet of Things: A Survey". *IEEE Internet of Things Journal* 3, Nr. 1 (February 2016): 70–95. https://doi.org/10.1109/JIOT.2015.2498900.
[92] Gaitan, Nicoleta-Cristina, Vasile Gheorghita Gaitan, Stefan Pentiuc, Ioan Ungurean, und E Dodiu. „Middleware Based Model of Heterogeneous Systems for SCADA Distributed Applications". *Advances in Electrical and Computer Engineering* 10 (1. Mai 2010). https://doi.org/10.4316/aece.2010.02021.
[93] Yin, S., und O. Kaynak. „Big Data for Modern Industry: Challenges and Trends [Point of View]". Proceedings of the IEEE 103, Nr. 2 (February 2015): 143–46. https://doi.org/10.1109/JPROC.2015.2388958.
[94] Kob, D. & Mayrhofer, R. Berg Huettenmaenn Monatsh (2018) 163: 253. https://doi.org/10.1007/s00501-018-0742-8
[95] Bauernhansl, Thomas. „Die Vierte Industrielle Revolution – Der Weg in ein wertschaffendes Produktionsparadigma". In Handbuch Industrie 4.0 Bd.4: Allgemeine Grundlagen, herausgegeben von Birgit Vogel-Heuser, Thomas Bauernhansl, und Michael ten Hompel, 1–31. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-53254-6_1.
[96] „Cyber-physical microservices: An IoT-based framework for manufacturing systems". 2018 IEEE Industrial Cyber-Physical Systems (ICPS), Industrial Cyber-Physical Systems (ICPS), 2018 IEEE, 2018, 232. https://doi.org/10.1109/ICPHYS.2018.8387665.
[97] Diedrich, Christian, und Matthias Riedl. „Integration von Automatisierungsgeräten in Industrie-4.0-Komponenten". In Handbuch Industrie 4.0 Bd.2: Automatisierung, herausgegeben von Birgit Vogel-Heuser, Thomas Bauernhansl, und Michael ten Hompel, 279–92. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-53248-5_63.
[98] Fernández-Caramés, Tiago M., Paula Fraga-Lamas, Manuel Suárez-Albela, und Manuel A. Díaz-Bouza. „A Fog Computing Based Cyber-Physical System for the Automation of Pipe-Related Tasks in the Industry 4.0 Shipyard". Sensors 18, Nr. 6 (Juni 2018): 1961. https://doi.org/10.3390/s18061961.
[99] García, Marcelo V., Edurne Irisarri, Federico Pérez, Elisabet Estévez, und Marga Marcos. „Arquitectura de Automatización basada en Sistemas Ciberfísicos para la Fabricación Flexible en la Industria de Petróleo y Gas". Revista Iberoamericana de Automática e Informática industrial 15, Nr. 2 (5. March 2018): 156–66. https://doi.org/10.4995/riai.2017.8823.
[100] Hofmann, Erik, und Marco Rüsch. „Industry 4.0 and the current status as well as future prospects on logistics". Computers in Industry 89 (1. August 2017): 23–34. https://doi.org/10.1016/j.compind.2017.04.002.
[101] Kagermann, Henning. „Chancen von Industrie 4.0 nutzen". In Handbuch Industrie 4.0 Bd.4: Allgemeine Grundlagen, herausgegeben von Birgit Vogel-Heuser, Thomas Bauernhansl, und Michael ten Hompel, 237–48. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-53254-6_12.
[102] Kayabay, K., M. O. Gökalp, P. E. Eren, und A. Koçyiğit. „[WiP] A Workflow and Cloud Based Service-Oriented Architecture for Distributed Manufacturing in Industry 4.0 Context". In 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA), 88–92, 2018. https://doi.org/10.1109/SOCA.2018.00020.

[103] Mayer, Felix, und Dorothea Pantförder. „Unterstützung des Menschen in Cyber-Physical Production Systems“. In Handbuch Industrie 4.0 Bd.2: Automatisierung, herausgegeben von Birgit Vogel-Heuser, Thomas Bauernhansl, und Michael ten Hompel, 525–35. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-53248-5_76.

[104] Nowakowski, E., M. Farwick, T. Trojer, M. Haeusler, J. Kessler, und R. Breu. „Enterprise Architecture Planning in the Context of Industry 4.0 Transformations“. In 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), 35–43, 2018. https://doi.org/10.1109/EDOC.2018.00015.

[105] Schöning, Harald, und Marc Dorchain. „Big Smart Data – Intelligent Operations, Analysis und Process Alignment“. In Handbuch Industrie 4.0 Bd.2: Automatisierung, herausgegeben von Birgit Vogel-Heuser, Thomas Bauernhansl, und Michael ten Hompel, 457–69. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-53248-5_70.

[106] Tauchnitz, Thomas. „Schnittstellen ermöglichen Datenintegration in der Prozessindustrie“. In Handbuch Industrie 4.0 Bd.2: Automatisierung, herausgegeben von Birgit Vogel-Heuser, Thomas Bauernhansl, und Michael ten Hompel, 335–48. Springer Reference Technik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-53248-5_64.

[107] Wolter, M.I., Mönnig, A., Hummel, M., Schneemann, C., Weber, E., Zika, G., Helmrich, R., Maier, T., Neuber-Pohl, C., n.d. Industrie 4.0 und die Folgen für Arbeitsmarkt und Wirtschaft: Szenariorechnungen im Rahmen der BIBB-IAB-Qualifikations- und Berufsfeldprojektionen 70.

A. Appendix 1 - Use Case Details

In section 3.1 two usage scenarios were defined, which contained abbreviated use cases. These use cases are fully defined in the sub-sections A.1and A.2 and include a full description, pre-conditions, post-conditions, and error-scenarios.

A.1 Corrosion lab

In this scenario it is assumed that the parts are devices which have two sensor values:

- Q-Factor (1 to 10 in steps of 1 or 0 if not defined)
- current position as a Zone A, B, C or undefined

For the scenario it does not matter if these sensors are stored permanently, implemented as real sensor values, or just computed on the fly by looking up the data in a back-end data source.

For the corrosion lab scenario (see section 3.1.1) Table A.1 defines the relevant use cases:

Table A.1: Corrosion lab scenario complete use case table

Use Case		

Get part location	Description:	Get the current zone (A, B, C or undefined) where the part (identified by its id) is currently located. This can be either the last transmitted position or the last stored value (needed after restart of gateway) or undefined in case no position is available.
	Pre-Condition:	Device-Id is valid and device is found During loading the last known position has to be restored as current value
	Post-Condition:	N/A
	Error Scenario:	Invalid device id causes an exception / error to be returned to the called
Query Q-factor for part	Description:	Get the current Q-factor as sensor data for the part (which is a device identified by its id). The default for new devices would be 0
	Pre-Condition:	Device-Id is valid and device is found
	Post-Condition:	N/A
	Error Scenario:	Invalid device id causes an exception / error to be returned to the called
Cyclic calculation of Q-factor	Description:	This use case is assumed to be run cyclic every 30 seconds to always have the most up to date factors available. In each run iterate over all defined parts (devices) and those which are currently valid (being in a defined zone) will have a Q-Factor (1 to 10 as value in steps of 1) assigned. Non-Valid parts or parts currently not in use will receive 0 automatically. The Q-factor will be determined for the usage scenario as a random number from 1 to 10 (in reality it would be a complex calculation for the parts movements in chambers, etc.)
	Pre-Condition:	There are defined parts

	Post-Condition:	All defined parts have a Q-factor associated (0 or 1..10)
	Error Scenario:	N/A
Send part movements in 3 zones and undefined	Description:	An external process (RFID simulator) to the gateway scans the gateway for all parts currently defined and for each part sends a random new position A, B, C or undefined every second. Therefore, all parts move every second into a new position (or remain at their current one if the new position equals the current one)
	Pre-Condition:	There are defined parts
	Post-Condition:	All defined parts have a new position after each iteration
	Error Scenario:	Parts in an invalid state are detected by the validation handler in the gateway and treated accordingly
Read chamber data	Description:	The chambers are simulated by an external process to the gateway which randomly generates values for the humidity (0..100%), temperature (233K to 333K[146]) and salt brine concentration (0..60%). Again by random either no data will be returned (to simulate sensor failure) or a value outside the defined range (to simulate reading fault). It should be possible to set an option that only erroneous or no data is generated for some time to force an error condition in the gateway. These chamber values are to be read by the gateway using a PULL operation every second and stored internally as the current readings for the chamber.

[146] 0°C = 273,5 K -> -40° .. +60° = 233K .. 333K

		In the validation handler each sensor is checked for the defined limits and in case a limit is breached the value is ignored
	Pre-Condition:	The external process is available for reading
	Post-Condition:	The provided value is taken as the new value for the devices gateway's sensor data
	Error Scenario:	External process is not available raises an alert Invalid data is checked by the validation handler and ignored
Check if part status is ok	Description:	This validation is taking place every time a part moves by means of the simulator sending a new part position. It is checked that: - When a part is in the wrong chamber for the selected test an alert is raised. To figure out the "right chamber" the part id can be used to look up the valid zone using a match table in the database or internal storage by means of SQL or Web-service call - When parts are moved out of a chamber into the pre-chamber-zone and not restored into a chamber within 15 seconds an alert is raised Check if the part is not in an invalid chamber for the current test, the part is not scheduled to be tested, the part has been tested already or the part has been "forgotten" in the preparation area
	Pre-Condition:	A part was moved and the simulator sent a new zone
	Post-Condition:	If everything is fine nothing happens, otherwise an alert is raised
	Error Scenario:	N/A
	Description:	This use case is assumed to be run cyclic every 10 seconds to always have a proper system state.

Check read cycle frequency		In each iteration it is checked, that chamber data has been read and is valid within the last 10 seconds. Otherwise an alert is raised.
	Pre-Condition:	N/A
	Post-Condition:	If everything is fine nothing happens, otherwise an alert is raised
	Error Scenario:	N/A
Raise alert	Description:	This use case is intended to be implemented as writing to an actuator, whereas the actuator would be "alerting service". Every time something is written to the actuator it is actually forwarded to an external process which will display the data written to the actuator on the console[147].
	Pre-Condition:	Actuator is registered External process is running so the actuator can forward the message
	Post-Condition:	N/A
	Error Scenario:	When the external process is not running an exception / error is returned to the caller

[147] In real life this might be sending information via SMS or E-Mail to a central instance, etc.

A.2 Exhibition visitor congestion display system

In this scenario it is assumed that the simulated users are sensors of a device "exhibition_users" and each sensor value represents the current position as X/Y tuple in the serialized form of a Point[148] class of the .NET runtime environment. The heatmap is a special sensor of the same device as well.

For the scenario no sensor values are stored to the database, so just kept in memory and initialized as empty points (X/Y = 0).

For the exhibition visitor congestion display system scenario (see section 3.1.2) Table A.2 defines the relevant use cases:

Table A.2: Exhibition visitor congestion display system scenario complete use case table

Use Case		
Send position	Description:	An external process to the gateway is assumed to scan all defined users (sensors) and generate movement changes as sensor data every second. The change is always in arbitrary random directions with random speeds (from 0 to 2 units per iteration). If a "wall" of the defined coordinate space (100 x 100) is hit, the direction is reversed. In random intervals wrong coordinates (too large and too small) are supposed to be sent to the gateway.
	Pre-Condition:	Users are defined as sensors of the device

[148] https://msdn.microsoft.com/library/system.drawing.point(v=vs.110).aspx

	Post-Condition:	The sensors (users) have a new coordinate
	Error Scenario:	N/A
Read heat map	Description:	Read the current value of sensor heatmap and return the data as is. If no data is present, calculate the heatmap on the fly (like the cycle task would do).
	Pre-Condition:	N/A
	Post-Condition:	N/A
	Error Scenario:	N/A
Validate position	Description:	Implemented as a validation rule for the new arriving position the X/Y coordinate are checked against the "walls" of the coordinate space. If a violation occurs the data change is ignored and the user is set to an undefined position (0,0)
	Pre-Condition:	Inbound sensor data with new position
	Post-Condition:	Valid data causes the sensor to update, invalid data is ignored and user is in undefined position
	Error Scenario:	N/A
Calculate heat map data	Description:	This is intended as a cyclic operation running every 1 second. All defined users having a valid position are traversed and the heatmap data is calculated. For the heatmap the coordinate space is divided in 10 units in X and Y direction (so in total 100 rectangles) and the number of users in each rectangle is stored in an array (for example [0, 0] -> 1 user, [1, 0] -> 4 users, …). This array is serialized and stored as the value of the "heatmap" sensor
	Pre-Condition:	Valid users are present
	Post-Condition:	Heatmap data is stored in sensor heatmap

	Error Scenario:	N/A

B. Appendix 2 - Specification Details

B.1 Gateway architecture feature definition

Table B.1: Gateway architecture feature definition

Feature / Functionality	Meaning / Importance
Customization required before use	The main question for many integration environments is if you have to adjust the underlying gateway before it can be used or is it usable "out of the box". For larger systems the customization is no issue as it usually is only a minor part of any project, yet for smaller scale projects the setup costs and knowledge involved can be a major factor. Especially the complexity which usually comes with highly customizable solutions is a challenge for many integrators as it requires a deep knowledge of all involved subsystems.
Available "in-house"	Can the gateway be used / operated within the premises of the integration environment or does it have to run in the "outside" world? Many use cases simply prohibit – either by means of security, cost, operational requirements or simply policy – the use of external systems for any kind of more sensitive data. Especially for more localized integration scenarios (machines, local sensors, etc.) where direct feedback from the gateway to the sensors and for example autonomous support is important this "in-house" availability is a required feature. External systems could not be reachable due to link problems, etc. – which might happen in-house as well – yet, the probability is higher as more externally controlled edges in the communication path are involved.
Hostable in the cloud	This feature is of paramount important when measurements of geographically disperse items are to be taken, which have no common network link to a centralized gateway system.

	Here a cloud-based solution is the only option. Usually this implies a real large-scale gateway where many devices are connected and data is exchanged all the time.
Amount of infrastructure / additional services needed	The higher the amount of additional infrastructure and service is, the more complex a solution usually gets – making it harder to change and enhance in the future as well. Every additional component adds insecurity and dependencies which in the long run can render a system very hard to adapt.
Cost	Cost usually is an important factor for decisions as it is a constant drain and usually occurs monthly (especially for cloud based solutions). The total cost of a solution is usually the combination of the fixed costs (either initially or monthly) and the running costs. The more complex a solution is – especially in terms of additional components / services – the costlier it usually ends to be. This usually stems from the fact that for each special component specialized knowledge for integration and maintenance must be present or purchased.
Support for very large number of devices	Usually a gateway will have to handle between several and in medium-sized environments 200 – 500 sensors. Yet in large and very-large scale environments this could increase to several thousand or even hundred thousand. In case such large device numbers are involved normally these devices are geographically separated and therefore cloud based gateways become more important as well. There exist special cases like RFID usage or wearable devices, where large numbers of devices can be reached in very small space and then this factor could become an issue for scaling and resilience of "in-house" solutions as well.
Failover, Cluster-Support	When only several devices are connected to a gateway in an experimental environment or in case a system failure is not a real problem then these features have no real meaning.

	Usually, as soon as gateways have to write sensor values, or an autonomous operation takes place, then these features become mandatory.
Data can be written to the sensors	Does the gateway support writing values to the devices, or is only read support provided? As most gateways are usually used to only read data (for writing often dedicated hardware solutions – PLCs – are used in the industry[149]) this feature is mostly relevant in no time-critical and no harmful processes. Yet in combination with PLCs it can be extremely valuable as the writing of a value (which would be done by the PLC) can be triggered by writing a value to the PLC. PLCs are usually networked (at least the modern ones) but have no or very little process integration capabilities (they are very technical devices). So the gateway can then use the native PLC protocol (mostly a fieldbus protocol) and access the PLC (submit a command) which is then resulting in a sensor write by the PLC. Therefore the gateway then acts as a mediator (protocol converter) between client and PLC (and from there the sensor)
Asynchronous information about changes	A client can either perform some kind of “busy polling” to always retrieve the latest value or is called back asynchronously when a change occurs. Busy polling as such is no problem as long as the frequency is not too high and not too many requests for too many devices arrive (causing load issues). The preferred approach for scalability and a more reactive integration pattern would be an asynchronous information based on which the client could perform some action.

[149] Especially as they guarantee either the writing to a sensor or the raising of an error condition if it cannot be written. When for example sending a “close” command to a valve in a chemical process it is of paramount importance to know that it is either closed within x msec or a high-priority alert is raised.

Autonomous system support / workflow exection	In case the gateway is supposed to perform independent operations (for example control some actuators based on some inbound readings or perform some general action whenever a value changes) autonomous operation of the gateway becomes important. In case the gateway offers no such support another system (typically a client listening to the triggering values) would perform the needed operations. Therefore, in the end the result might be the same, yet another (virtual) machine is needed, more integration has to be done and more failures can occur.
Data is stored internally	To be able to provide a client with the ability to query historic data for a value the gateway must store the data in a way that it is accessible. This usually requires the data to be stored internally (or at least close) to the gateway. As usually historic data is of relevance for consumers (at least in business process integration nearly always someone has to create a kind of report with historic data), this is a rather important feature.
Inbound data can data be intercepted, analysed, checked	In case data cannot be intercepted, analysed and checked in the gateway this functionality has to be implemented on a separate system. Yet here the problem might be that the perhaps incorrect value already is “in the system” and might be consumed by another data consumer with the incorrect value. In addition, if the value has to be dismissed entirely, the history records might have been written already, etc. Thus, for a real business integration, where data is supposed to be manipulated in the system which generates it, this requirement is of paramount importance as it forms the backbone of the data enrichment for processes. To enrich data, it is necessary to align and normalize the data first.
Support for data querying	As it is extremely important for consumers to be able to retrieve data, which was captured by a gateway in an efficient and

	functional way each gateway architecture should offer some functionality to do so.
Communication pattern	In principle any communication pattern is usable. Yet from the author's experience it has proven that most integrators have more problems with message based systems as the pure asynchronous operation – especially when writing is involved – can be a major issue regarding the learning curve and constant source of errors and problems. Most integrators are simply used to the pattern to call an action in a gateway and receive an answer right away. A transaction which might or might not succeed is therefore a problem. Especially when it is important to know if a for example write was performed (which is not easily possible with message based systems)
Synchronous / asynchronous communication model	This is the same area as the communication pattern.
Level of coupling of gateway and consumers (client)	In case a coupling is loose this might be a benefit (or even a requirement) in geographically separated environments as there no constant link can be easily established and maintained. Thus it is important that no permanent (tight) connection is needed. Yet in a more local environment this tightness using a permanent connection has benefits as well. Usually data can be much faster transferred, less errors occur and the integration very often is much smoother. For pure loosely coupled systems message exchange is the only real communication pattern.
Preservation of state (knowledge about a sensor)	In case the gateway is simply used as a message relay, the maintenance of state is no big issue. Yet as soon as the current sensor state gets more important (like for example in autonomous system integration) the gateway has to have a state representation of the sensor. In addition if no state is preserved no querying of a "current" state can be done which means that either the consuming application

	has to maintain the state for the relevant data of the sensors or the data has to be queried live all the time.
Semantic data value increase	Does the gateway allow new data to be generated or existing data enhanced in a way that new business value can be generated?

C. Appendix 3 - Device-Details

In this appendix details about devices covered by the thesis (section C.1), common operations in device integration (section C.2) and integration patterns of devices into business process (section C.3) are given.

C.1 Devices considered for the DBGA

In this thesis two kinds of devices are considered:

- **Physical devices** with some communication facilities which exist as an entity in the real world;
- **Logical (virtual) devices** which only exist as a definition, data source or provider of information towards the architecture.

For physical devices it does not matter if a device is a simple temperature sensor with just 2 pins and a signal as an analog voltage which has to be read via an A/D converter behind an interface port, or a complex device like a smartphone which can communicate using TCP/IP and WS* services or REST, or a PLC with a Profibus interface.

The very same applies to logical devices – these can be simple discrete values or complex ones generated by internal rules. The value which is generated serves as data input for the consumer as well.

In higher abstraction layers, physical and logical devices start to blend in their characteristics as they share functionality, communication patterns, and so on. Due to the, at least currently, ever-increasing communication and CPU / memory facilities of physical devices (according to [84], [87]), quite soon many physical devices will have no difference to a logical device whatsoever[150].

The DBGA has to be agnostic to the fact if it is a physical or logical device – operations and proceedings must be the same for both.

[150] Here IoT would be different as devices have form-factors, communication and energy requirements, etc. which are not easy to overcome despite the general advances in computer technology

C.1.1 Physical devices

Physical devices considered in this thesis can be divided into **network-enabled** devices which have support for networking operation (with the corresponding protocol stack), and **non-networked** devices, which are usually connected by means of some wire-protocol (RS-232, RS-485, I2C, SPI, FireWire, USB, etc.). A third group would be **RFID-based** devices which can be interfaced using RFID technology.

C.1.2 Networked devices

A typical example of a networked device (in industry environments) would be a Siemens S7[151] PLC, which provides sensors and actuators, and is accessed using a TCP/IP based protocol in specialized libraries.

Other very important devices nowadays would be smartphones or tablets. They are ubiquitous and pervasive, provide a plethora of sensors[152], are accessible directly via TCP/IP (using wireless communication technology), provide displays for interaction and have enormous processing power and long energy support. Due to their form factor they can remain with the user and thus provide very personalized sensing and feedback (actuator support) which is increasingly interesting in business-process integration – mainly as user interface for process interaction[153] [84].

In general, for most use-cases the networked device (especially with TCP/IP) will be the "device of choice" as it is easy to connect and most operational parameters are standardized (by various specifications).

C.1.3 Non-networked devices

These devices usually are connected by means of some serial protocol using wires or being read from an I/O port. Classical examples would be SPI bus

---

[151] https://w3.siemens.com/mcms/programmable-logic-controller/en/advanced-controller/s7-300/pages/default.aspx

[152] The current iPhone 6 provides: 3 axis gyroscope, 3 axis accelerometer, digital compass, iBeacon, proximity sensor, ambient light sensor, touch id fingerprint reader, barometer, camera

[153] http://www.criticalmanufacturing.com/de/newsroom/blog/posts/blog/mobile-devices-in-industry-4-0#.Wy-DC6dKjD8

(see[40]) or I2C (see [39]) as de facto industry standards or RS-232 and RS-485 as more "old-fashioned" and much less standardized connection methods.

The big difference between "old connection styles" (like RS-232) and new ones is that the old ones in principle only defined the physical layer (the hardware layout, pins, voltages) and gave no or very limited protocol specification, timings, etc.[154].

The lack of standardization / definition especially can lead to many problems when connecting these devices, as any protocol to transport data over these connections (including timing) must be specially implemented and is usually supplier-proprietary. A very typical example of a RS-232 protocol would be 3964R from Siemens which was used before the Ethernet was available for automation in almost all PLC applications undertaken by Siemens and others.

Nowadays – using SPI bus and I2C – these problems are solved and the interfacing is usually very simple as most runtime environments or development systems contain standard libraries / functionality to handle the device access. So usually to read a value from an I2C device just means calling some library function.

One additional challenge is that I2C and SPI bus were designed to connect sensors using this serial protocol to a system, yet a PC usually is not equipped for a task like that (missing I/O boards). So, to handle this efficiently, usually a device board, which contains independent processing power, is placed in between the PC and the I2C / SMI bus sensor. Boards like this would then act, as mentioned before, as network enabled devices or custom devices like for example a smartphone which internally uses I2C very often to connect the various sensors.

C.1.3.1 RFID-based devices

These devices are special, since to interface with them dedicated hardware is required to generate the radio frequencies that are necessary. Thus, device interaction is usually done via an RFID-gateway or transponder.

---

[154] So it was more like "connect to the device using 300 bauds on a RS-232 cross-cable"

The RFID-based device is mostly used like an indicator (thus like a flag) telling a receiver (and therefore the gateway ultimately) that the device (and the thing it is attached to) is in its vicinity. For this a RFID-tag (which is the most common use case with RFID) is programmed with an identifier by which the labelled thing can be identified further on[155]. By means of triangulation and signal run length calculation the 3-dimensional position can be calculated and delivered. Most use cases involving RFID-sensors just check if a device is in or out of a boundary (for example surgical instruments in or out of an operation theatre) or if the device moves between "cells" (like in the author's customer case) or is idling.

C.1.4 Logical devices

A logical device is anything which just exists as a virtual concept – an abstraction – capable to submit and / or receive data. So, a logical device can be an input (sensor), an output (actuator), or both. Therefore, in principle a logical device can be considered a software service which can be interfaced using standard protocols and integrated and consumed instantly into a business process, by means of pre-aggregated data and processing by pre-defined rules and operations the logical device.

A logical device, like any physical device, has a number of properties which would be its sensors and perhaps actuators as well.

For the DBGA, logical devices are very important as they provide an abstraction layer which allows chaining of devices. A logical device can be the input (together with other logical / physical devices) for another output (again as logical device).

Typical examples of logical devices would be:

- A process which monitors current factory-wide counters (using a variety of data sources) and publishes them (for example production count, error count, etc.) as sensor readings to the gateway;

[155] Usually tags have only approximately 18 bytes of usable data, so not very can be fitted on them

- A process which takes output from the gateway (for example some process indicators) and publishes them towards a company-wide data warehouse so the data can be further aggregated.

Such a typical combination and chaining is shown in Figure C.1 where one logical and one physical device are combined by means for a process which acts as the source for device X (in the gateway) with the sensor values associated.

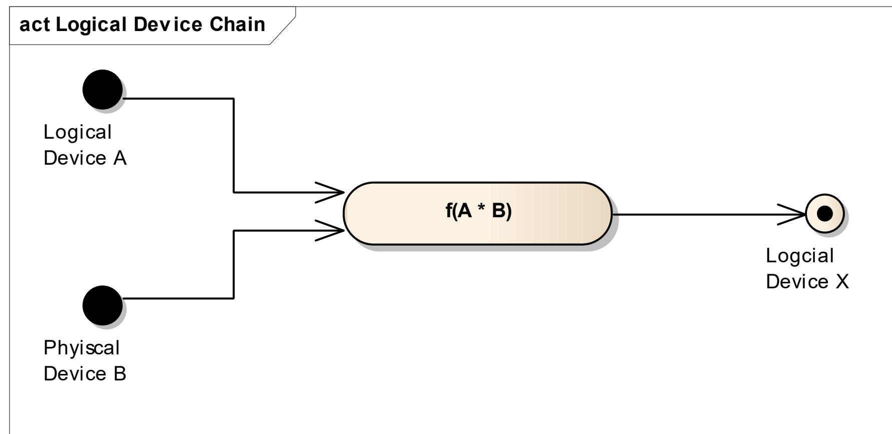

Figure C.1: Logical Device Chain

C.1.5 Consumers of device data considered

Device data can be principally consumed by any software capable to request the data and process the response. In this thesis it is assumed that data consumers of device data fall into 3 different categories (which share similarities, of course):

1) Business processes in general which receive data from devices when data changes or a scanning cycle retrieves new values.

   Most data will be delivered as pushing data to the process so it can react accordingly, instead of actively (busy) poll for data from the process.

2) "Classical" applications interested in just some data items – in principle the same as business processes, yet more often scanning for data (polling) will be involved.

3) Business (or information) users querying data from the device using methods like ODATA to retrieve more structured information on a

large-scale basis. Usually these users will create reports, etc. with the data.

Here only active data retrieval (query) is relevant.

C.2 Common operations in device integration

When working with devices, regardless of the integration method used, the operations described in the sub-sections of this section (section B to section C.2.5) are commonplace and exist almost everywhere.

C.2.1 Reading device data (including cleansing, filtering and manipulating)

When values are to be consumed, any application – gateways as well – have to obey the following sequence for any data item entering the system if they want to preserve data integrity:

- Read data item;
- Check that the data:
  - is valid as such (data format, precision) – syntax analysis;
  - is within the defined bounds (min / max) – band pass filter;
  - fits / matches the previous values (sudden surges, drops);
  - is continuous (if a sequencing is possible);
- Handle "out-of-sync" situations (data items missing, data items sent several times, timestamp mismatch so older data is sent after newer data).

Handling these requirements can be especially non-trivial if the data rate is accelerating and data is to be handled in a uniform, yet extendable way (so new formats are to be integrated, new checks done). In addition, additional issues will arise when business process relevant filtering has to be applied. If for example, every new value has to be checked whether it matches the current production batch limits, then these limits must be retrieved dynamically and the check done as well.

One of the most crucial problems here is how erroneous data is to be handled – should it be ignored and thus losing values, should a default be applied, should the wrong data item (with a flag set, that it is wrong) be stored? This all depends on the consuming application's requirements and must be addressed

in a device gateway (or the consuming application if data is to be retrieved directly).

When using a device gateway, the consumer should be relieved of these considerations and be able to entirely focus on the task to process the arriving data, assuming that it is valid – or at least to be able to distinguish easily that it is not.

In general handling inbound device data by any system should follow predefined workflows for the data analysis and manipulation so that it can be formally checked, evaluated and quality controlled.

C.2.2 Storing, caching and querying of device data

Aside from just plain reading, which often is just protocol conversion, filtering and cleansing (see section B) many implementation scenarios have in addition a requirement to access (query) historical device data.

If an integration technology therefore has no provision to do exactly this, that part has to be implemented by the consumer, which in case of several consumers of the same data might require:

- That each consumer stores its own data (thus creating data silos with all the problems about security, access, etc.);
- A centralized instance is created which stores the data and consumers than access that instance to retrieve historical data.

In the case where historical data is realized by an integration technology caching that data – or at least keeping something like a least recently used cache – is mandatory so that not each request (which very often is around the current data) has to be served from external data pools.

If historical data is provisioned another challenge to be solved by the integration technology is how data can be queried. This could take several options:

- Provide a mechanism in the integration technology by means of a query language, a query facility, and so on;
- Provide access to an underlying known data source like a SQL database and provide views therein to access the data;

- Support a data query interface like ODATA or R, which especially for statistical queries, is very powerful[156].

C.2.3 Informing consumers about change of data

As more and more devices will participate in business processes (especially with the advent of smart mobile devices), scanning values from these devices will be not an option for consumers in many environments. Simply the sheer amount and the availability and accessibility are not given or cannot be guaranteed, so it has to be possible that data changes are sent to consumers whenever they happen. This can take the form that the device only signals when it has changed data or that devices are scanned by a device gateway and only changed data is propagated further to consumers.

Either way requires a changed behavior from consumers as they have to start working in a passive mode – being triggered by changed data and then doing something, as opposed to the typical usual model of scanning for data and then handling the change.

Currently this change information or better notification can be – depending on available technology – be sent by using:

- communication from the gateway to the consumer in a bi-directional way which requires a permanent connection between the two;
- writing data into a queue where a consumer is either notified by the queue client-side instance, or the consumer scans the queue client-side (which is much faster and less intrusive to a central system);
- the server sends asynchronous notifications using technologies like Web-Sockets or SignalR[157].

C.2.4 Writing data to actuators

Writing data to devices is a non-trivial task as it often involves quite complex error handling and especially recovery operations. What should be done when a write failed (and how the sender is informed – even more important for

---

[156] Used in for example sMAP

[157] SignalR is a Microsoft technology (http://www.asp.net/signalr) using Web Sockets or alternative protocols for older browsers which allows a server to call back clients

asynchronous operations)? Or partially failed (for example write has to occur on 4 ports where 3 can be written and the 4th generates an interrupt)?

Write operations might need to be either synchronous (as it might take some time (longer than a typical accepted response time) to actually trigger the change, etc.) or involves dedicated sub-systems which are not available at the moment. Should several requests exist at the same time to change a single actuator either some decision is required as to who will win. Options – which should be configurable - are: last one wins, first one wins, or all writes have to be done in the order of their arrival.

In cases where caches for data are involved these caches have to be synchronized with the values to be written as well – thus always the current value which is present at the device should be the internal value as well.

So, writing in general has to be either straightforward and thus “write-through” by design (only supporting direct writing in synchronous manner) or the whole implications of the above-mentioned issues have to be catered for with queues for the values to be written and so on.

C.2.5 Device and operations control (supervision)

Very often when dealing with devices, especially in slightly more complex scenarios than just reading some values, it becomes important that either data changes, data arrives, data is written, and so on. Usually these events are time-related (for example a part must change its state in 15 minutes or a sensor value must arrive every max. 30 sec) and therefore supervision has to be time-related as well. As each sensor is unique, it is important, that thresholds can be set individually for each sensor and centralized tasks – the watchdog-timers, as they are timer-based – control the conformity of the current environment to the definition.

C.3 Device to business processes integration and integration patterns

Today most companies utilize business processes at various levels in addition to technical processes usually happening in industry automation (for instance SCADA-system to control material flows, etc.). These business processes can be as simple as procedural definitions of what has to be done when and how,

or more complex forms where data is collected, enriched and then forwarded to a subsequent system in a potentially fully automated way[158].

Reasons to have these business processes in place are:

- Obtain a better understanding of the operations a company performs and especially their relationships and interactions. If the business processes are formally defined understanding of them is greatly improved;
- As a side effect especially, the interactions between processes are usually important as here “media breaks”, “organizational breaks”, etc. happen, which often causes problems;
- If the activities and interactions / relationships are defined, stakeholders can communicate efficiently and start analyzing and improving the performance of these processes;
- Flexibility as the ability to change. Having business processes implemented means that change can be evaluated before it is implemented, its results measured objectively and very often implemented without causing too much disruption as well.
  A side effect and result of the flexibility is perhaps the most important reason:
- The ability to continuously improve processes based on new evidence and knowledge.

A typical example would be a pre-production quality assurance process (part approval process) which guarantees that only parts which comply with a given specification / requirement are later used during the production process. In Figure C.2 this quality assurance process is shown in an UML activity diagram.

---

[158] Often using orchestration languages such as BPML (Business Process Markup Language)

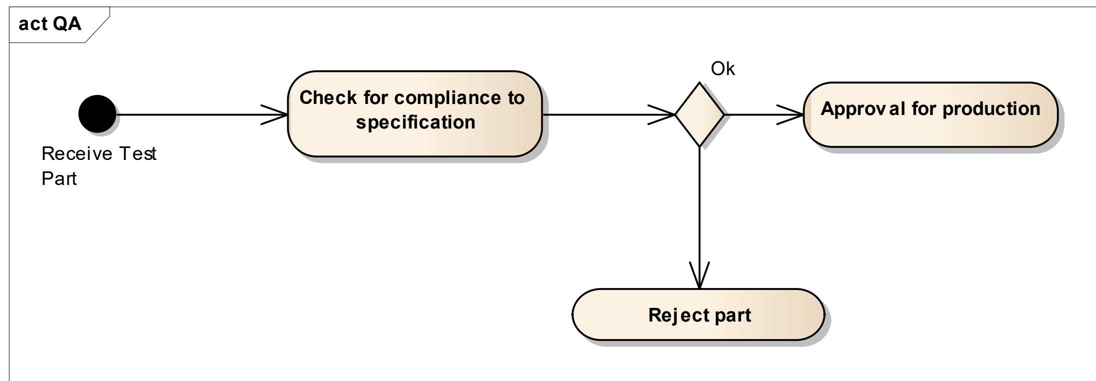

Figure C.2: QA process

Currently (at the author's customer) this process is handled manually by operators whereas the decision “if ok or not” should be really an automated one. The needed measures are clearly identifiable, the requirements (as being a norm), are too. So, the process as such would simply have to take the evaluated measure and compare it to the limit and using the result make a decision including the generation of the approval report (at least to some extent).

Historically devices (regardless if sensor or actuator) of any kind were direct communicants to / for industrial automation, but not business processes, as they were considered too technical, less advanced to communicate easily, and so on. Therefore, any integration of devices into business processes was done by (very often manually) pre-aggregated (offline) data which was stored somewhere (often in a proprietary format), but no feedback ever was given directly back to the devices. So, there was almost never an interaction between the business process and the device reading data and even less sending data, therefore not really integrating the devices.

As established and automated business processes are an absolute necessity for any improvement as envisioned by Industry 4.0. [87] the business processes need outside communication either to send or receive information in becoming part of an ecosystem of information exchange. This interaction exchange is vital for the overall use of the business processes as only then (when information exchange happens) real “flows” between processes can take place.

To achieve this, these “interfaces” or “connection points” where the interaction happens, need special care as they have to be easy to integrate into

the business process as well as easily accessible for making the business processes integration smooth, straightforward and efficient.

Still the old problem remains that the business processes usually does not consider the devices in their flows for several reasons:

- Device integration is troublesome business (error handling, recovery, reliability, …);
- Usually you have to deal with several (sometimes even very different ones) of these devices (at the same time);
- Various communication patterns, protocols and methods;
- Very often information is very "low" level and not aggregated / accumulated.

Yet this is only the communication part. Other areas worth much deeper investigation are:

- the pre-aggregation and cleansing of data to act as better information sources for the business process (if data is always valid and even pre-aggregated than internal operations are simpler and easier);

- the provisioning of value added data so that more complex / efficient decisions can be taken (like for example the costs involved with a particular measurement like the power consumption);

- the storage of historical data to be able to quickly retrieve them in case of need (to quickly evaluate if this pattern happened before and how);

- the ability to have autonomous functionality as a kind of "sub business process" or agent within the larger business process directly on or near the device (so the overall process becomes simpler as sub-activities can be off-loaded);

- Service oriented architecture (SOA[159]) is very well understood in business process integration. Why not have the device gateway as a

---

[159] SOA-Definition: http://www.opengroup.org/soa/source-book/soa/soa.htm#soa_definition

service which exposes its underlying devices / information in a similar way?

- business processes usually possess a different structure than technical processes (for instance communication patterns, solutions involved, technologies used, etc.) this implies that the device integration must obey these as well.

All these points together are the basis and motivation to investigate a new device-gateway architecture to provide answers and solutions to these points.

When it comes to integrate devices (either directly or via intermediaries) the following two patterns exist[160], which are further elaborated in the following sub-sections of this section:

- Point-to-point
- Hub-and-Spoke integration

C.3.1 Point-to-Point integration

A typical representation of point-to-point integration can be found in Figure C.3. Each node interested in some interaction with a device gateway addresses the gateway directly and communicates with it. In case a direct integration is used then each node would address (if physically possible as a device might be attached to a node and not accessible from other nodes then) each device required.

[160] It should be noted that regardless of the integration pattern used, direct or intermediary based integration can be used

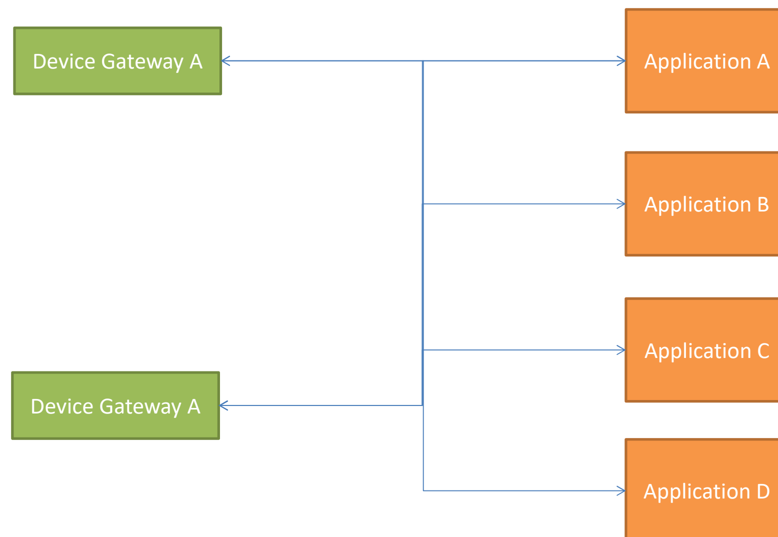

Figure C.3: Point-to-Point integration

In a classical enterprise application integration scenario many people would argue that this tends to generate many links between systems and each change in one system has to be propagated to all other systems connected with it (meaning that some implementation must be changed, etc.). Therefore, after some time the ability to change (which is vital for any business as otherwise processes cannot be changed and standstill occurs) is lessened and lessened and costs spiral as integration is expensive. For other people this is acceptable – as long as the number of integrated systems is rather low, as alternative patterns (see section C.3.2) have a huge initial overhead upfront and can be rather complex as well.

Normally in a device-gateway context the problem is much less exaggerated as usually only one gateway exists which changes very seldom and the attached devices usually have a very slow change cycle as well. So, change in general will be low and if there is a change it usually is only the data transferred, not the protocol as such. When new value types or new measures have to be consumed by the receiving application this implies some change anyhow (except if the application is prepared to handle new data automatically by discovery). The same is true for writing data to actuators.

Therefore, the point-to-point integration is the usual integration pattern successfully used by most people without too many problems. In general, it produces stable, easy to control systems – as long as:

- Not too many devices are connected
  and / or
- Not too many consumers have to be addressed

When talking about large numbers of devices or consumers for business process integration then the situation starts to change as here normal device-gateways and direct integration will usually not be able to cope with the load and not scale well enough.

Classical examples for such kind of integration scenarios would be:

- power consumption meters of a whole suburb transmit their data to a centralized place so distribution and power-control can be optimized
- smart devices send data for further processing to a central location
- traffic congestion monitoring devices are queried by thousands of people all the time and especially when automated updates are offered this causes huge loads

Giving "many" a more absolute value is not easily possible but a rough indicator would be 5,000 - 10,000 as indicated by tests performed in this thesis and the author's experience. More can be handled as peak, but not on a sustained basis. When it comes to signaling changes back to consumers the number drops significantly as the network overhead and time required has to be considered as well.

As device numbers grow over time (just considering the amount of RFID-tags) and integration will get more important for business process scaling options for device gateways have to be revised and investigated to cope with the increase in communication.

C.3.2 Hub-and-Spoke integration

An alternative to the direct integration is the hub-and-spoke integration which is shown in Figure C.4. Here all nodes are connected as spokes to the centralized hub.

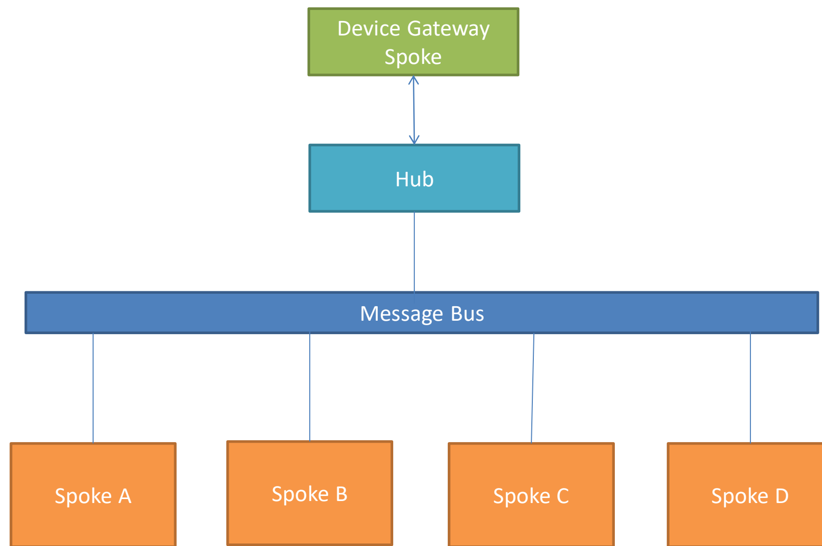

Figure C.4: Hub-and-Spoke integration

The device-gateway spoke (which can be either a direct integrated one or one using an intermediary) sends any data to the hub which then, by analyzing message structure and content[161] and inspecting any given rule-set, discovers the intended receiving spokes. These will then be either notified or the message will be placed in their message store to be retrieved later on.

This pattern is mainly employed in cases where inbound data has to be distributed directly to a (potentially large) set of receivers and very often no further querying of data takes place[162]; so in principle data streaming from producer to consumer.

Large scale business processes involving large number of the same kind of devices often use this technology – and nowadays very often smart devices – which requires a rather extensive infrastructure to be operational[163].

The biggest benefits of this integration pattern are:

---

[161] Usually messages are transmitted as XML so corresponding techniques like XPath, XQuery, etc. are used

[162] Depending on the technology used the message could be transformed and enriched on the way by means of filters / modifies on the bus like for example in Azure IoT

[163] Hub servers are usually clustered as well as the database to store the messages

- A change in the device communication or intermediary usually will not change the messages transported
- Messages usually are designed for change and downward compatible (so in new messages the newer parts will be simply ignored or not even sent by the hub)
- As it is entirely rule-based receiver definition is easy to change
- Receivers only have to consume messages – no interaction with a gateway, etc.
- Message delivery is guaranteed – any message being received will, as long as the receiver gets online, be delivered

As much as this pattern is optimized for large consumer numbers it has some problems as well:

- Message receiver spoke identification is very CPU intense (XML analysis, rule sets, transforms, etc.) and thus the hub is usually highly utilized in CPU, so load-distribution, clustered system, etc. have to be considered
- Due to the message format the amount of data to be transported is very high compared to the real payload. It is not uncommon to transmit 200 or more bytes in XML for a simple 2-byte value which explains ratios (net data to control data) of 1:10 to 1:100 very often seen in real-world environments
- All messages not being consumed immediately have to be stored (or discarded by rules) which makes maintenance more difficult.
- The later retrieval of "old" messages, which might be outdated, is another challenge which must be compensated by TTL[164] definitions
- Interaction with the device is normally limited to receiving new values (whenever they arrive – no direct control there) and sending changes to be sent to the device (no control either – everything is asynchronous).

Still a hub-and-spoke integration, given the right scenario and infrastructure, can be a big improvement. So any device gateway should consider these points

---

[164] TTL = Time To Live -> the amount of time a message / data items is remaining active before it is discarded

as well and apply especially methodologies for serving large consumer sets in a straightforward manner.

D. Appendix 4 – Design Details

In this appendix details about specific design issues are given. Areas covered are the data model (section D.1), data consumers (section D.2), communication (section D.3) and process (dynamic) view details (section D.4).

D.1 Data Model

The current data model of the DBGA is shown in Figure D.1

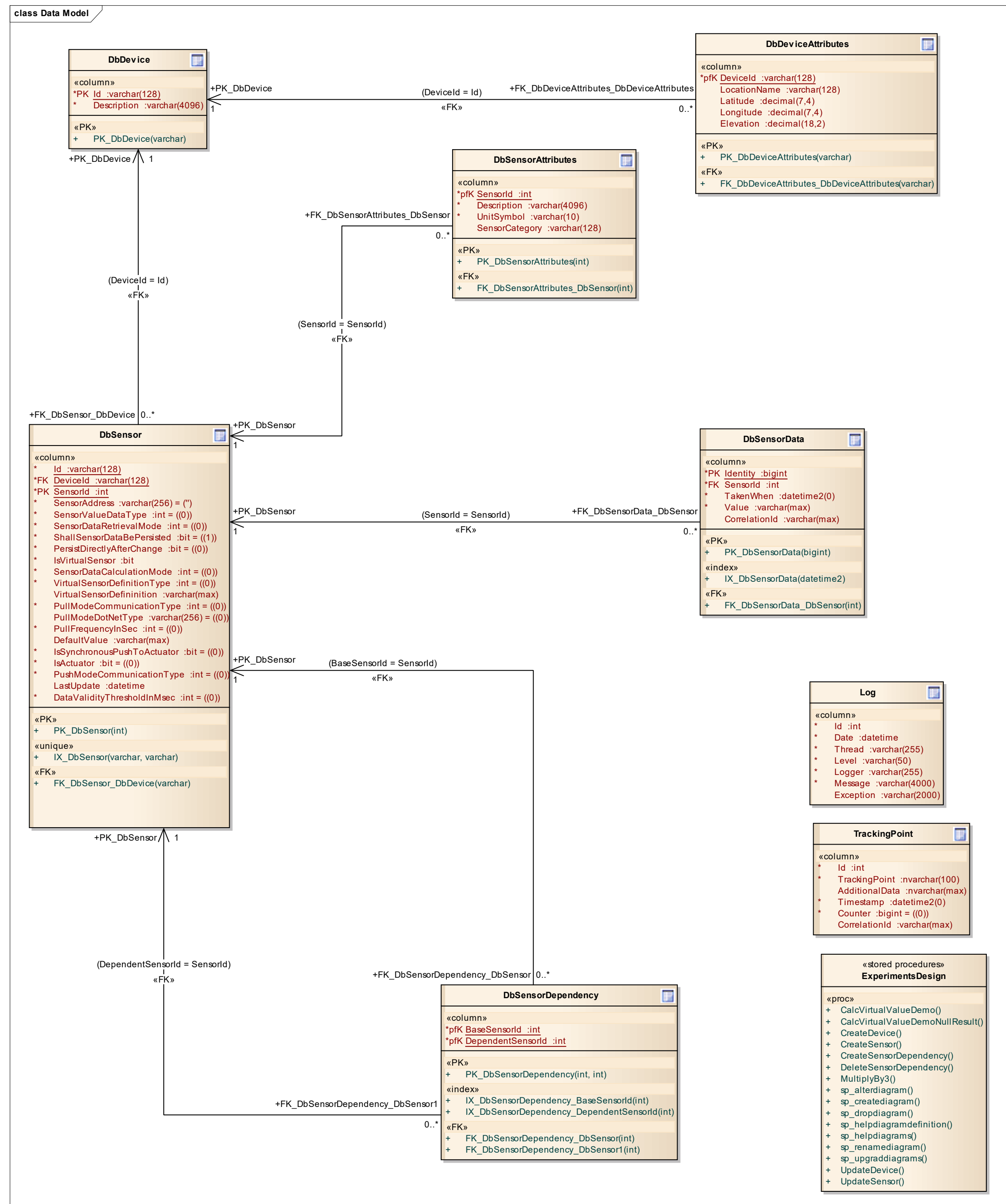

Figure D.1: Data Model

The current design can be described like this:

- Devices, sensors and their relation to each other (who is dependent upon whom) are defined by the three tables *DbDevice*, *DbSensor* and *DbSensorDependency*. Here all information necessary to handle these instances is stored.

  In *DbDevice* the actual device with an IP address, etc. is defined.
  DbSensor defines the sensor (or actuator) on a given device with additional attributes like the scanning frequency, the rules how a virtual value is computed, etc.).
  DbSensorDependency describes as a 1:n relation which sensor depends on which sensor(s) thus the ability to form these dependency trees (see Data entity trees (section 4.1.3) for more details)

- As a core assumption (based on the characteristic Data format agnostic (section 3.5.9)) was to store all data values as simple strings in the database the table *DbSensorData* holds the values of persisted sensor values over time[165].

  To allow for workflow synchronization an optional correlation id can be stored with the data item as well, so that especially long-running software-agents can synchronize on this correlation id

- The *TrackingPoint* is mainly present for runtime measurement of the systems health and performance and consists of entries generated dynamically by the device gateway core

- Log is the central system log where all centralized logging happens

---

[165] Storing data as simple strings seems like a waste of lots of resources, yet the evaluation tests in chapter 6 (especially test 4 in section 6.5) show clearly that this is no issue. Data has to be converted anyhow back and forth for transmission over communication protocols and as REST and SOAP use textual representation to store the data as plain text saves at least one conversion.

In the following sub-sections, the individual tables of the data model (see the general description in section 4.7) are described:

D.1.1 DbDevice

Here the device, which contains the sensors (for virtual sensors this is a virtual device) is described. As written before the meta-data associated with a device is rather sparse in the device gateway architecture, but based on projects needs easily extendable. To do so the table *DbDeviceAttributes* is designed, which is a simple container for the device attributes.

The primary key was chosen to be a VARCHAR(128) as this allows for easy readable and symbolic device-names (like MACHINE12_CONTROLBOX2) to be used instead of numbers.

D.1.2 DbDeviceAttributes

This is the main place to store attributes about a device, especially in consideration how to find and retrieve it during a discovery process.

Initially the location and X/Y/Z-coordinates are stored here to allow geographical identification (for example using Google Maps).

D.1.3 DbSensor

The sensor definition is the core of the whole device gateway as here many fields are controlling the actual flow of tasks and processes. These fields (or attributes) can be found in Table D.1:

Table D.1: Sensor definition attributes

Attribute	Description
SensorAddress	Every sensor has an address (could be origin or destination as well) which is in a syntax and format depending on the designated protocol to use. For IPv4 this would be for example 192.168.0.1 whereas for a virtual value, which represents a system runtime value this could be "percentage" to indicate which value is needed

SensorValueDataType	Describes the original data type so semantic checks can be performed for any inbound string representation
SensorDataRetrievalMode	Describes how a value gets to the system (PUSH / PULL / both) or in case it is an actuator how data is written to the actuator (PushOnChange)
ShallSensorDataBePersisted	Flag, if data after a change shall be written to the data store or only kept internally online. For for example only temporary values (or very fast changing ones) this option is relevant as it limits the traffic to the database considerably
PersistDirectlyAfterChange	If changed data is to be persisted this flag controls if data is cached and persisted in bulk operations (many updates in one transaction), or if each update has to occur at the very moment of change. In case it is not entirely necessary to synchronously update the bulk update (asynchronously) is much better for overall system performance
IsVirtualSensor	Flag, if this sensor is a virtual one
VirtualSensorDataCalculationMode	How / when the sensor data is calculated (by a cyclic background task, on change of an underlying (dependent upon) data item, on request of the value)
VirtualSensorDefinitionType	How the evaluation is to be done (Formula, C#-Expression, F#-

	Expression[166], IronPython[167], JavaScript, Typename of .NET-object[168], SQL Stored Procedure)
VirtualSensorDefininition	The actual script or expression to use for evaluation. In case of IronPython or JavaScript the actual script is stored here
PullModeCommunicationHandler	Defines the type of the communication handler to use for PULL operations
PullFrequencyInMsec	Every how many msec shall a pull occur
DefaultValue	The default value (as string) this sensor reports if queried without valid data present
IsSynchronousPushToActuator	Is a write to an actuator necessary directly after updating the value, or can the write request be queued
IsActuator	Is the sensor an actuator?
PushModeCommunicationType	Defines the type of the communication handler to use for PULL operations
LastUpdate	When was the last update of the current value
DataValidityThresholdInMsec	How many msec since the last update is a sensor to be still considered valid, before a new evaluation has to take place

[166] The .NET-runtime, which is used for the reference implementation allows code to be compiled "on the fly" and then executed. The provided C# or F# (for functional programming) script is then compiled at runtime and executed. A similar technology would be available with Java and most other languages as well

[167] Porting of the Python runtime environment to .NET

[168] Again this is specific to the .NET runtime environment and a valid type name would for example be System.Int32. Similar technology exists in Java, too.

D.1.4 DbSensorAttributes

Sensor attributes are mainly used for finding sensors and at the moment limited to a description, a free text category and a unit symbol which can be helpful in conversions. In case of need this area can be easily extended to allow for many more attributes.

D.1.5 DbSensorData

Sensor data which is stored in the database consists of the sensor-id it belongs to, the value as a character representation and a timestamp when the value was taken. The timestamp reflects the time the value was received by the system, not when it was written, as this - due to caching and asynchronous writing - could be much later.

In case a workflow is involved very often an optional correlation id is needed so that long running processes can define data belonging to them.

D.1.6 DbSensorDependency

In a sensor dependency it is simply described which sensor is dependent upon which. Any cyclic references (A => B, B => C, C => A) have to be resolved by higher logical layers. The database simply stores any data passed down.

D.1.7 TrackingPoint

The TrackingPoints are used by various part of the device gateway to track current operations and allow the basis for further performance and system maintenance analysis. As an example, tracking points are written when new values arrive, during the processing of the value change and after storing. This way it can be later analysed which operations take which time, how the system load was, etc.

In general, a tracking point can be used by any component (custom ones, including communication handlers as well) and store timestamps, counters, additional data, etc.

D.1.8 Log

All exceptions, errors and problems are protocolled (alongside runtime information and debugging messages) into the system wide log. The level defines if it is an error, a warning, info or a debug message.

D.2 Data consumers for the device gateway

Classical consumers for the device gateway would be any kind of business process with an interest in the data of the devices / sensors. It does not matter if physical or virtual data is relevant – either is treated uniformly and delivered to the receiver.

So, the consumer in general is only interested in a speedy, reliable and as optimal for the integration scenario as possible way, to retrieve the necessary data. Especially when historical data is being retrieved the ability to access these in a straightforward and logical way makes a big difference.

As indicated the most classical consumer of the gateway would be a business process of some kind, which further utilizes the data to create additional information. Yet, it could be (and will be more and more in the future) a web site (even facebook.com might be an option here as values can be just an embedded information) which uses some of the values (physical / virtual) provided to visualize data or inform users about a critical status.

In general, in the past there was a clear focus on B2B[169] integration while in the future the B2C[170] and M2M[171] will gain significant importance. This implies that for B2C, human readable data and ease of use is important, whereas in M2M the clearly structured message (which could be automatically analyzed and optionally converted) is paramount, which implies different protocols and data volumes for both approaches as well [48].

So, the gateway design has to cater for the specific demands of the following user groups:

- Business processes
- Workflows
- Web sites (even facebook.com might be an option here as values can be just an embedded information into a site); therefore, mobile users of any kind
- "Broadcasting" services like message queuing to publish changes

---

[169] B2B = Business to Business

[170] B2C = Business to Consumer

[171] M2M = Machine to Machine

The big difference between these different user groups is that some usually react to changes in terms of being triggered (by for example workflows or queues), yet others query the gateway permanently to retrieve values (web sites with busy polling might be the best example).

As these requests could be towards highly chained virtual values, where the execution might take a lot of CPU cycles, the gateway must take considerable care not to be overwhelmed by these requests. Classical approaches to cope with these request "bursts" would be appropriate caching or request quota handling so that the impact can be easier levied.

At the moment the current gateway design only considers the caching approach because quotas require knowledge about the consumer on the other side as each quota might be different. As this implies additional requirements (like usage identification) caching was the simpler and more straightforward approach.

D.3 DBGA communication handler details

The current logical overview of the communication handler structure can be seen in Figure D.2.

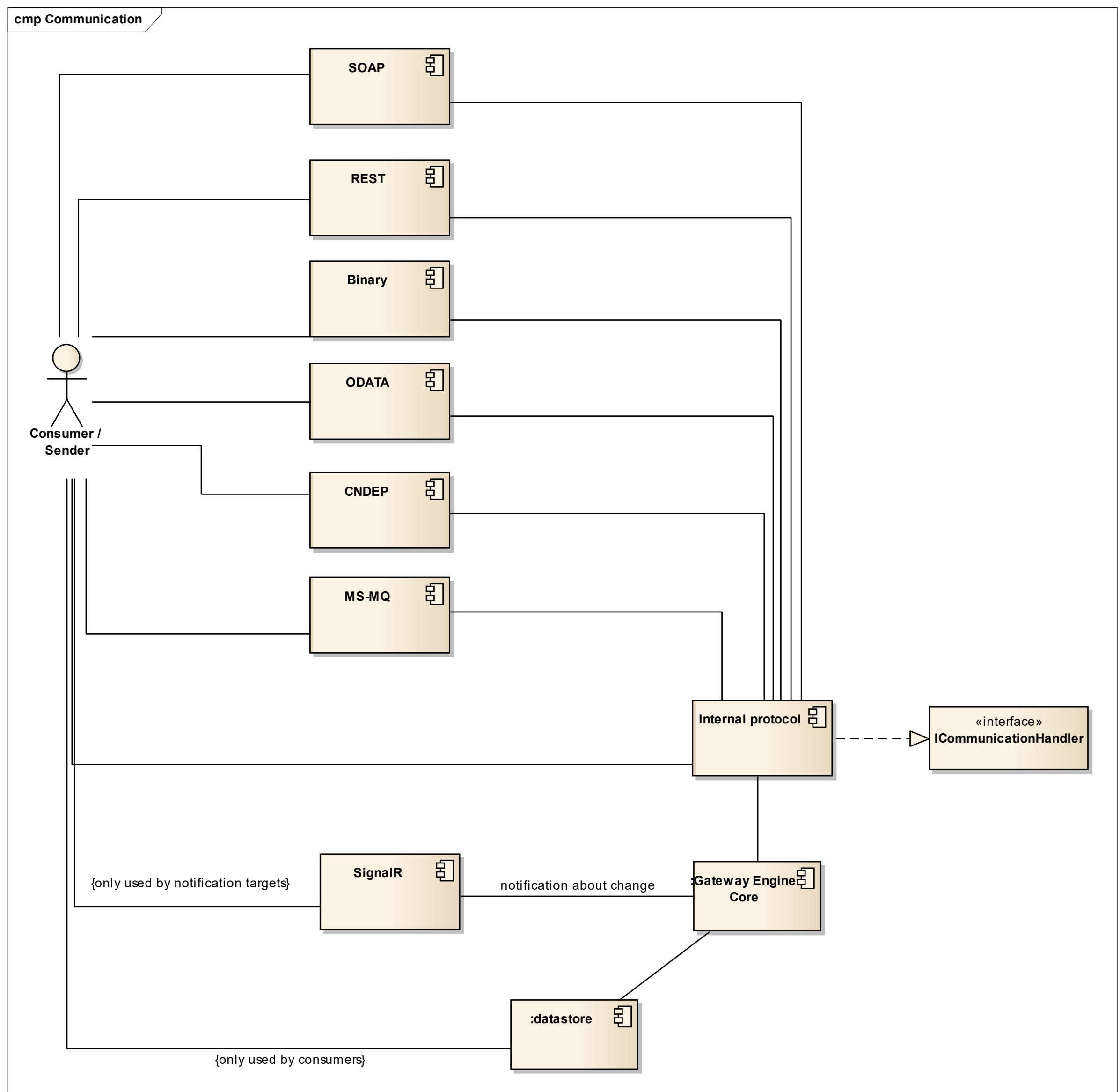

Figure D.2: Communication interfaces

Each communication handler has specific use cases, which can be briefly characterized like:

- SOAP

  In this message API style ([66]) a common set of related messages, which is not bound to a specific procedure, is used to invoke the desired action. These messages usually are grouped in three different types: Command, Event and Document messages, where all three are used in the gateway architecture. In a message all relevant information for the specific operation is contained and delivered as well, therefore the message acts as a container (or envelope) for the enclosed data.
  A major advantage of message style APIs - and especially the SOAP variety - is a very good support for asynchronous operations, which

helps to provide blocking.

- REST
  In this resource API style ([66]) all data is addressed as resources, where in terms of the device gateway a resource would be a device, a sensor or a specific reading of a sensor value. The relevant resources are represented using the URI during the communication process with HTTP as communication protocol. A typical URI to address a sensor would then be: *http://devicegateway/sensors/3/A12345* which would address sensor A12345 on device 3, etc.
  The state of these resources is then manipulated through representations, which is the basis for the definition of the term "REST" (Representational State Transfer). The device gateway adheres to the concept of "RESTfulness" (implementing REST), yet it should be noted, that not every resource API can be considered "RESTful".
  The gateway uses the standard HTTP methods (which are required to be considered RESTful as well):
    - PUT (create or update resources)
    - GET (retrieve - here optionally additional parameters can be passed like the timeframe, etc.)
    - DELETE to remove a resource
    - POST to set a new value for a sensor

As a special notion POST is not idempotent (which sometimes is the case with resource APIs), so every time POST is called, a new value is considered to be written for a sensor.

- Binary: Similar to SOAP, just the message exchange is done using binary data instead of XML which is much faster and as no parsing of XML is involved much less CPU consuming as well
- CNDEP: An experimental protocol designed primarily to quickly push data into the gateway. Not very efficient and elegant when used for data retrieval, thus mostly used by sensors to write data
- ODATA: a semantic layer on top of REST providing standardized services for querying and potentially updating data – mainly used together with more general purpose consumer like Microsoft Excel or Business Intelligence Applications as it allows for a more general

approach. As stated in the ODATA specification ([46]): "The OData Protocol is different from other REST-based web service approaches in that it provides a uniform way to describe both the data and the data model. This improves semantic interoperability between systems and allows an ecosystem to emerge"

- Message-Queue based: message queuing interfaces imply that data access is not online and synchronous. Requests are sent to the server and responses delivered asynchronously. This is especially interesting in either long-running requests (for example deliver the average of the last 500 values for 1000 sensors) or in large amount of changes being sent which need no instant (online) processing, but can be processed at leisure by the gateway. Such a scenario would be for example relevant when it is more important to keep the history of a data entity value over time than the actual value at a given moment of time. Message queuing usually will be mainly used by consumers

D.4 Process (Dynamic) view details

In the following sub-sections various details of the process (dynamic) view which is discussed in section 4.5 can be found.

D.4.1 Receiver tasks for data endpoint issued writes

For the defined protocol handles for inbound write requests, each protocol handler implies its own semantics:

- Message based API (for example SOAP) will be triggered by an external request and then the web service behind the SOAP request will handle it.
- Resource based API (for example REST) will be triggered by an external request as well and then the passed in data (usually POST for write operations) is process.
- Message Queue based protocols require to actively read a message queue, retrieve the packet and then process it which implies that the "active" side is within the receiver task

Message and resource-based API styles tasks (with ODATA as a special case of REST) as well as binary receiver tasks are usually passive and hosted within a

receiver application. This application will activate the corresponding task upon arrival of a new request and processing starts.

If for example the Internet Information Server (IIS) under Windows is used for handling these kind of requests (using WCF for it), it is a matter of defining the service and the binding and any inbound message to this service / port is directed towards it, where the processing happens. The same is true for REST protocol handling as here the resulting HTTP handler is simply registered within IIS for the port. Regardless whether IIS or an independent hosting process is used (could be from Apache or an own implementation) for these kinds of tasks the following statements are still true:

- Activation is done from the outside by a request arriving
- Tasks are stateless as they - in the extreme - could be removed / recycled after each processing of a request
- The hosting process cares usually very efficiently for load balancing, instantiation of new tasks and in general resource management

Considering these points, a design decision was made to use as much of existing system infrastructure as possible and design the receiver tasks in a way that they just plug into what is present.

This implies:

- For REST and ODATA a http handler which plugs into the http processing engine (IIS in the reference implementation)
- For SOAP a http handler and for binary a binary request handler which plugs into the http processing engine (WCF under IIS in the reference implementation[172])
- For Message Queueing and CNDEP an independent process who scans the configured message queues and CNDEP channels and processes incoming requests

As CNDEP and Message Queue container have to be provided by an implementation the design here was straightforward to have simple processes which just instantiate the handler class and then use blocking reads for newly

---

[172] In WCF the service contract is the same for both - just the binding would be (by configuration file) different for SOAP and binary protocol

arriving data packets. In this way no resources are used until a real new request arrives and if it arrives it can be handled instantaneously. Should more requests arrive than can be handled they will be either queued (message queue) or, after the TCP-buffer for CNDEP is exceeded discarded. Should this happen the sender must take care to properly react on the return codes of the requests and re-submit the data items.

D.4.2 Callback handling

For the various callback types the UML class diagram can be seen in Figure D.3:

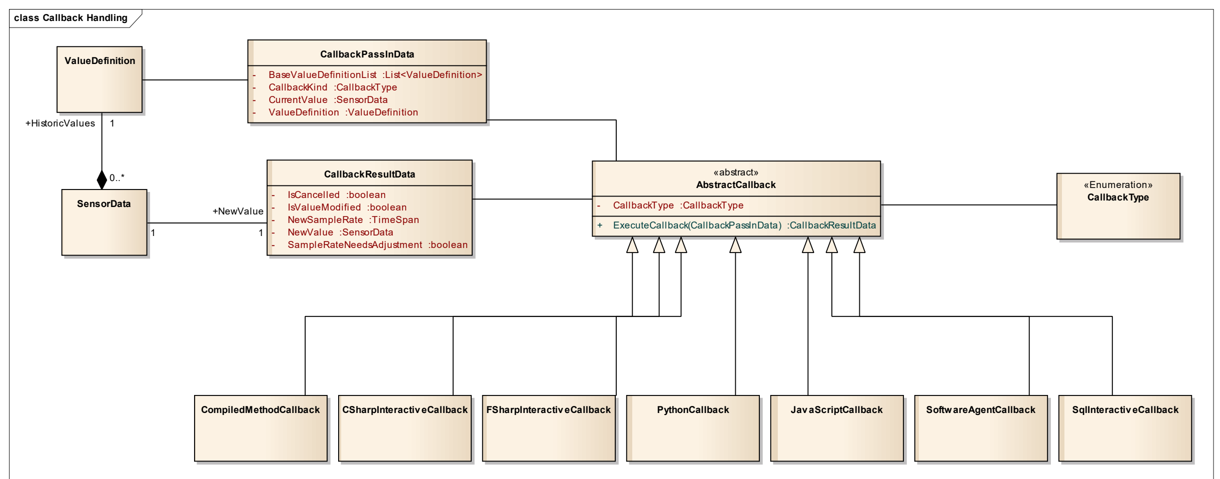

Figure D.3: Callback types

There is an abstract base class *AbstractCallback* which defines some properties - most importantly the type this callback is intended for. By specifying the type of callback, the usage of the callback is defined as well. Currently one callback is assumed to handle only one callback type, yet this might be changed in the future so that one callback can handle several types.

A callback has to implement only the method *ExecuteCallback()* which gets passed in *CallbackPassInData* which describes the current value, its definition and the base definitions, which make up the current value. This is very important for example for virtual values as then all nodes in the tree can be traversed accordingly.

Via the pass in data, the callback has access to the currently cached (and non-cached by means of data store retrieval) historic values which might be needed to for example calculate a running mean.

The callback can signal to the initiator the next action by using the fields of the returned *CallbackResultData* object. Here it can be defined that the current operation shall be cancelled (*IsCancelled*) or the sample rates should be adjusted (*SampleRateNeedsAdjustment* together with *NewSampleRate*). When the value has been changed (for example in a *GeneralCheck* callback), to multiply every value by 2 for a compensation in a process, the new value is simply returned, or in case of a "normal" operation nothing happens at all, so the new value is simple the existing value.

A callback can be implemented using a variety of technologies to best suit the integration approach (and knowledge of the integrator). Table D.2 defines the current callback-kinds:

Table D.2: Callback kinds

Callback-Kind	Meaning
CompiledMethodCallback	A compiled method[173] will be executed
CSharpInteractiveCallback	For .NET the ability to execute a piece of plain-text C# code by on-the-fly compilation and execution[174]
FSharpInteractiveCallback	The same like with C# in F# which is more suited to functional programming and analysis
PythonCallback	On the fly execution of Python[175]
JavaScriptCallback	On the fly execution of JavaScript[176]
SqlInteractiveCallback	Execute code stored in a stored procedure in a SQL database
SoftwareAgentCallback	Execute a Workflow.

[173] in an Assembly in the .NET environment or a JAR in Java

[174] Something similar would be possible in Java as well

[175] IronPython.NET

[176] Hosted V8-engine (https://github.com/JavascriptNet/Javascript.Net)

D.4.2.1 Callback example in Python

As an example, a Python script (as a *GeneralCheck* callback) to multiply the provided value by 3 (to for example compensate for something) would look like:

```
import clr
import ValueManagement.DynamicCallback
from ValueManagement.DynamicCallback import *
clr.AddReference('GlobalDataContracts')
import GlobalDataContracts
from GlobalDataContracts import *

# callback for general check
def GeneralCheckCallback(callbackPassIn):
    callbackResult = CallbackResultData()
    callbackResult.IsValueModified = True
    callbackResult.NewValue =
      GlobalDataContracts.SensorData(str(int(callbackPas
      sIn.CurrentValue.Value) * 3))

return(callbackResult);
```

D.4.3 Error- and Log handling component

For an autonomous system like a device gateway, which usually just keeps running without any human interference for days or even weeks or months, it is of utmost importance to keep track of anything which goes wrong, as well as let someone know about problems.

To handle the error, runtime information and debug logging, an existing component log4Net from the Apache Software Foundation is used[177], which can be easily configured and covers all needed aspects. Log data is stored in a configurable location - for the reference implementation inside the database.

By using a provided appender (class *SmtpAppender*) output can be directly forwarded to an SMTP-gateway and therefore sent as an e-mail to a receiver. As the actions are highly configurable only errors of a certain level will be sent.

Using just standard components enables a very swift and fast development cycle with a low learning curve, yet they have to be replaced, should the device

[177] https://logging.apache.org/log4net/

gateway be ported to a different runtime environment. In case Java, this would be log4J[178] which is 100% similar to the used version.

D.4.4 Gateway engine core

The gateway engine core component acts as the central component which will launch all other operations and provides a complete repository of all loaded instances, services, etc. Thus, it is a central dictionary that any component inside the device gateway ecosystem can use to query the current environment and reach any other component registered.

The gateway engine will be usually launched from a launching process which is for example the device gateway server starter process. This launching process acts as a start-up routine, the main work is done inside the engine core.

D.4.5 Dynamic configuration

Configuration management is an important issue for a system like the device gateway yet the standard mechanisms provided by the hosting environments are enough to set the necessary values[179].

The design assumes a centralized Singleton which holds all configurable values as a centralized dictionary, acting as a cache as well. Writing is not permitted and an exception is generated as no write-through to the underlying data store can be guaranteed. Therefore, changing the configuration has to happen outside of the framework by writing into the textual configuration files using an editor.

By using the singleton approach the implementation can be easily changed should the need arise later on to a database-based version. Then only some methods have to be changed (fill the dictionary and write).

---

[178] https://logging.apache.org/log4j/2.x/

[179] As currently roughly 25 entries are configurable this seemed like a moderate approach