Jukka Asikainen

OPC UA Java History Gateway with Inherent Database Integration

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 2.4.2014

Thesis supervisor:

D.Sc.(Tech.) Ilkka Seilonen

Thesis advisor:

M.Sc.(Tech.) Jouni Aro

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY SCHOOL OF ELECTRICAL ENGINEERING	ABSTRACT OF THE MASTER'S THESIS

Author: Jukka Asikainen

Title: OPC UA Java History Gateway with Inherent Database Integration

Date: 2.4.2014 Language: English Number of pages:7+62

Department of Automation and Systems Technology

Professorship: Information and Computer Systems in Automation Code: AS-116

Supervisor: D.Sc.(Tech.) Ilkka Seilonen

Advisor: M.Sc.(Tech.) Jouni Aro

OPC Unified Automation is a highly-developed information modelling and managing framework in use in the automation industry. OPC UA takes into account the communication, data modelling and security aspects w.r.t information exchange between devices in the factory floor. In practice the information exchange is done between server and client instances. Of these, servers hold the process data, and clients access it. An intermediate gateway is developed, which accesses (and allows managing) several other servers from a single instance.

The storing of process data within the OPC UA framework is another main topic of this thesis. The thesis presents an SQL data model to storing time series process data acquired from multiple servers. The data itself is modelled using the OPC UA semantics. Additionally, the connectivity and data mapping to few SQL implementations is solved.

A solution addressing both the storing and integration aspects is introduced in the form of the OPC UA History Gateway. The OPC UA History Gateway illustrates capabilities of the OPC UA framework in the data acquisition and device integration in the modern automation environment. The implemented (prototype) solution is shown to aggregate and store plant floor device information. The OPC UA History Gateway also provides trend data to clients, making more refined data analysis possible.

Keywords: OPC UA Gateway, Java, SQL, OPC UA History Data

AALTO-YLIOPISTO	DIPLOMITYÖN
SÄHKOTEKNIIKAN KORKEAKOULU	TIIVISTELMÄ

Tekijä: Jukka Asikainen

Työn nimi: OPC UA Java Historia-Gateway ja tietokantaintegraatio

Päivämäärä: 2.4.2014 Kieli: Englanti Sivumäärä:7+62

Automaatio- ja systeemitekniikan laitos

Professuuri: Automaation tietotekniikka ja -järjestelmät Koodi: AS-116

Valvoja: TkT. Ilkka Seilonen

Ohjaaja: DI Jouni Aro

OPC Unified Automation on automaatioteollisuudessa käytetty määrittely tiedon mallintamiseen ja hallintaan. Määrittely kuvaa mm. kommunikoinnin, tiedon mallintamisen ja turvallisen tiedonsiirron automaatiolaitteiden välillä. Tiedonsiirto tapahtuu palvelimien ja näihin yhteydessä olevien asiakassovellusten välillä. Työssä toteutetaan eräänlainen välityspalvelin (gateway) tiedon kokoamiseen. Välityspalvelin on yhteydessä useisiin muihin palvelimiin, joiden tietoja voi käsitellä sen kautta.

Työn toinen tärkeä osa-alue on prosessitiedon tallentaminen SQL tietokantaan. Tähän tarkoitukseen työssä esitellään tietokannan rakenne, joka mahdollistaa OPC UA:lla mallinnetun tiedon tallentamisen (ja palauttamisen). Työssä myös toteutetaan prosessitiedon tallennus ja luku tietokannasta Javaa ja OPC UA:ta käyttäen. Samalla ratkaistaan osittainen OPC UA tietomallin esittäminen SQL tietokannassa.

Työn ratkaisuna on prototyyppi OPC UA historiavälityspalvelimesta. Palvelin kokoaa useiden OPC UA palvelimien tietoja yhteen palvelimeen ja kykenee näin yhdistämään laitetietoja laajalta alalta. Palvelin myös tarjoaa aikasarjoja (pysyvästi) tallennetusta prosessidatasta, mikä mahdollistaa kehittyneemmän tiedon analysoinnin.

Avainsanat: OPC UA välityspalvelin, OPC UA historiadata, Java, SQL

Preface

This thesis is fruit of long labour in the Helsinki University of Technology, lately part of the Aalto University under the name of School of Electrical Engineering. Dawning of a day, which contains my completed Master's Thesis, has not been self-evident throughout the years.

Thus, for the completion of this thesis, I want to thank my instructor Jouni Aro for guidance, advice and the vast amount of mentoring needed in the completion of this thesis. I want also to thank all the colleagues in Prosys PMS Ltd for creating friendly and easy-going atmosphere to work in (bad jokes included, for which I may also be held responsible), and for the ideas regarding this thesis. My gratitude goes also to Ilkka Seilonen, my supervisor, for helpful advice, comments and requirements on the thesis in progress and in the final stages. Finally, I want to thank mother for everlasting support and encouragement.

The most general precept I tried arduously to keep in mind through all of the writing of this thesis, is a quote by certain Ludwig Wittgenstein: 'Everything that can be said can be said clearly'. I can only hope to have achieved a tiny fraction of this in the following pages.

Otaniemi, 2.4.2014

Jukka Asikainen

Contents

Symbols and Abbreviations

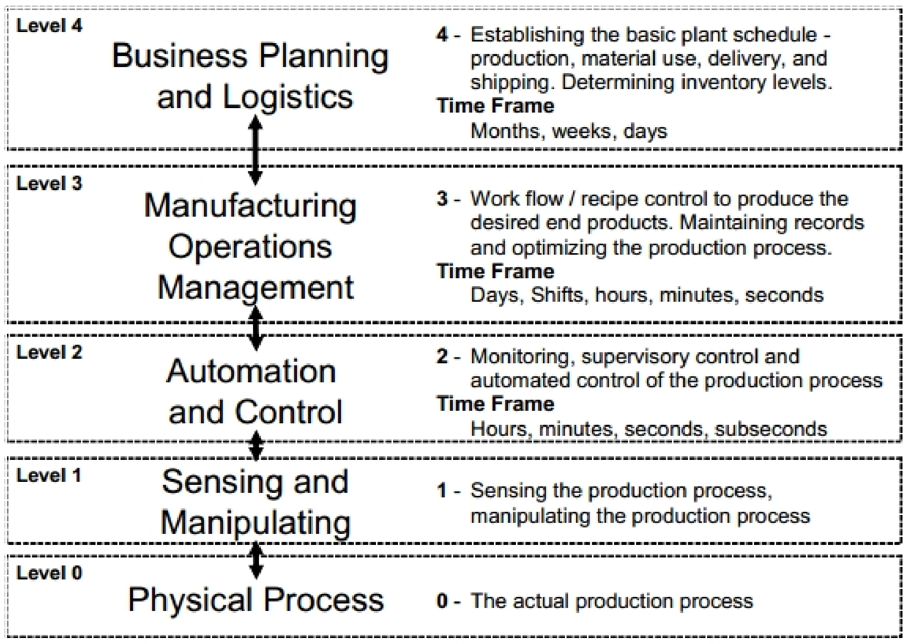
Abbreviations

A&E	Alarms & Events
ANSI	American National Standards Institute
API	Application Programming Interface
COM	Component Object Model
DA	Data Access
DBMS	Database Management Systems
DCOM	Distributed Component Object Model
DDL	Data Definition Language
DML	Data Manipulation Language
ER	Entity-Relationship
ERP	Enterprise Resource Planning
HA	High Availability
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
ISA	International Society of Automation
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
MES	Manufacturing Execution System
ODBMS	Object Database Management Systems
OLE	Object Linking and Embedding
OPC	OLE for Process Control
O/RM	Object-Relational Mapping
PCMS	Process Control Monitoring System
PLC	Programmable Logic Controller
RDBMS	Relational Database Management Systems
RM/V2	Relational Model: Version 2
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SCADA	Supervisory Control and Data Acquisition
SDK	Service Development Kit
SQL	Structured Query Language
TCP	Transmission Control Protocol
UA	Unified Architecture
UML	Unified Modelling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

1 Introduction

1.1 Background

The flow of information between layers of automation devices is vital to the effective management and control of the modern industrial environment. As the data that automation devices gather from their environment increases, so does the need to integrate this information into higher levels of the plant management infrastructure. With the rise of the manufacturing management systems (e.g. ERP (Enterprise Resource Planning), MES (Manufacturing Execution Systems), and SCADA (Supervisory Control and Data Acquisition) etc.), this information can be effectively used to maintain a real-time status of plant activities, ultimately enabling decision-making processes based on accurate data.



Level 4
Business Planning and Logistics
4 - Establishing the basic plant schedule - production, material use, delivery, and shipping. Determining inventory levels.
Time Frame
Months, weeks, days
Level 3
Manufacturing Operations Management
3 - Work flow / recipe control to produce the desired end products. Maintaining records and optimizing the production process.
Time Frame
Days, Shifts, hours, minutes, seconds
Level 2
Automation and Control
2 - Monitoring, supervisory control and automated control of the production process
Time Frame
Hours, minutes, seconds, subseconds
Level 1
Sensing and Manipulating
1 - Sensing the production process, manipulating the production process
Level 0
Physical Process
0 - The actual production process

Figure 1: The hierarchical representation of layers in a typical industrial scheme [1, p. 334]

Figure 1 illustrates the typical hierarchy and operations of the aforementioned industrial environment. [2] In the schema the environment is divided into five levels: Level 0 represents the physical reality in which the process operates. Level 1 defines available methods to affect and observe that process. Level 2 defines the control of the process and is highly intertwined with level one. Levels 1 and 2 together usually form the automation system, which is controlled by PLC (Programmable Logic Controller) and SCADA systems. Level 3 contains the coordination of the production resources at the scale of single items and is often managed by MES. Finally, Level 4 describes business functions used to capitalize the end product. ERP systems are designed to handle operations at this level.

With the help of the OPC UA (OPC Unified Architecture; details and definition of the framework are introduced in Section 2), the environment can be modelled more extensively in the lower levels of the hierarchy, and the management of the industrial plant in much finer detail becomes feasible. Thus integration of the levels from two upwards (cf. Figure 1) is eased. OPC UA is a powerful information modelling framework: it handles communication between devices, contains a highly developed semantic information model and acts as a specification to building software. The communication is handled with specialized service calls in the same manner as in modern Web Services, using modern network protocols.

The integration of the industrial environment is handled in practice with software such as OPC UA History Gateway. Gateways in general gather automation device data into a single container, which can be easily accessed from other parts of the organization hierarchy. Gateways can be also used to make legacy systems part of the current information infrastructure by wrapping them into up-to-date containers. Main part of this thesis reflects upon the implementation of the OPC UA History Gateway software using Java. The results of this thesis show the viability of the OPC UA approach in realizing device integration.

After the information is modelled, it is imperative to store this information in a permanent manner, especially if any kind of process history is needed. Thus the second main theme of this thesis is SQL (Structured Query Language) databases and their integration into the OPC UA. In many cases, the process variables and their values persist (or are needed to persist) in some kind of a database. It is beneficial to synchronize data from the logic controller level with this database as early as possible, at the same time minimizing errors in the transfer process. SQL was selected for its predominant status in the current DBMS (Database Management Systems) market, even though alternatives exist. In any case, databases are essential to managing any even modestly large industrial process, and the integration of the device data (in useful manner) into existing project management systems is often a costly and large operation. This thesis illustrates general database integration implemented in the lower levels of the organizational hierarchy.

1.2 Objectives

The main objective of this thesis was to develop a prototype of the OPC UA History Gateway which contains an integrated database functionality, showing the viability of the OPC UA framework in industrial device integration. To fulfil the purpose of the History Gateway, service calls in the OPC UA framework were needed to be relayed. Thus the operation of services within the OPC UA framework was one of the most essential topics of this thesis.

The objective of storing data into an SQL database using object-oriented programming language (such as Java) also provided its own difficulty. Thus one objective was the evaluation of the technologies solving the SQL connectivity. Use of SQL required a functional and efficient SQL database structure to be solved, along with all the implementation details of the selected technologies.

As the main objective of the work was to create functional software, an optimal design of the software needed to be devised. The design process itself was kept informal in order to be as effective as possible. Also, the existing OPC UA Java SDK (Service Development Kit) was taken as a starting point for the design and provided a mature framework into which incorporate the solution. This also minimized the need for extensive design paradigms.

Considerations presented above can be condensed into the following research questions, ordered from problem definitions to solving them in practice:

1) What are the difficulties in relaying service calls in the OPC UA framework?
2) What are the difficulties in integrating industrial process data into an SQL database using OPC UA framework and Java?
3) What is a suitable software structure to accommodate the relaying of service calls and SQL database integration
4) What is a suitable SQL database structure for storing data modelled in the OPC UA framework?
5) How can the relaying of service calls and SQL database integration be implemented in Java?

1.3 Research Methods

The main research methods in the writing and implementation part of the thesis have been literature review and few common computer science tools. More specifically, the central references have been the OPC Unified Architecture by Mahnke et al. and The Relational Model for Database Management: Version 2 by E. F. Codd, along with SQL standards. [3, 4, 5, 6]

Of the computer science tools, UML (Unified Modelling Language) diagrams were used in the design of the software. UML diagrams also structured the development process by providing a concrete scope of all the needed changes. ER (Entity-Relationship) diagrams were used in the design of the database structure. Microsoft based tools including Visio and SQL Server Management Studio were used to create the illustrations.

Other viable sources of information have been numerous conversations with fellow employees in Prosys PMS Ltd and all the source code and documentation available in the Prosys Java SDK. Also, the many-faceted process of creating a functional software provided many insights into the OPC UA and SQL, main frameworks of the thesis.

1.4 Structure of Work

This work is divided into seven main structures. In the first section the concepts, scope and motivation of the work are introduced together with some necessary background information. The second and third sections explore the theoretical backgrounds of the OPC UA and SQL frameworks, respectively. These sections lay the foundation from which the research questions can be answered. The fourth section compares few vendors of SQL based database management systems. The fifth section presents possibilities to solving the database connectivity in the Java environment. These two sections define and answer research questions regarding the difficulties in SQL integration. In the sixth section the implemented solution is finally presented along with some practical issues and notes. This section also contains definite answers to most research questions. Lastly, the seventh section contains the summary and discussion of the obtained results.

2 OPC Unified Architecture

2.1 Introduction and History

OPC UA (OPC Unified Architecture) is the latest development of the OPC (OLE for Process Control) standard. This standard is widely used in the communication between different entities and layers of industrial automation devices. The main philosophy is based on the well-known client-server paradigm (developed in Xerox PARC as early as 70's [7, p. 297]), in which servers contain the data and clients access this data by reading and writing operations. For example, a PLC program can write its data into an OPC UA server, from which the data can then be read by any OPC UA client positioned further in the organizational hierarchy (cf. Figure 1).

Historically, OPC UA is the most recent development of an older OPC standard also used as a communication interface for automation devices. The fundamental difference between these standards lies in the communication protocol used: the traditional OPC standard is based on the Microsoft developed OLE (Object Linking and Embedding) protocols, originally designed to allow the exchange of objects between different software. This technology then evolved into the Microsoft COM (Component Object Model) and DCOM (Distributed Component Object Model), still quite early (mid 90's) Microsoft approaches to sharing and binding to components and objects. OPC UA, for one, uses modern network protocols such as TCP or HTTP/SOAP as its communication protocol.

The traditional OPC is an aggregate of several standards taking into account different needs of the end user, containing such components as the OPC DA (Data Access) and OPC A&E (Alarms & Events). As these standards were created separately within a several year time span, the end result was somewhat patchwork collection of features, which also forced the separate development of essential features for the OPC client and server software. Even more shackling is the limitation to the COM and DCOM communication, as the latter is the only way of transmitting data over networks and is irreparably outdated with the modern network protocols and firewalls, especially in the domain of the network security.

OPC UA on the other hand is (as the name suggests) a unified approach to both the data modelling and communication, encompassing the relevant aspects of the industrial automation device handling. In more general terms, OPC UA can be seen as a change to SOA (Service-Oriented Architecture), a software design pattern used in the implementation of the modern Web Services, among other things. SOA emphasizes the use of separate, self-contained services, which contain no service calls directly into each other. Instead, services use defined protocols with metadata-laden information to pass and receive messages. Such an architecture is ideal for a modular data exchange between programs and parts of programs, also easing the implementation of the security features between these programs. [8, 9]

2.2 OPC UA System Architecture

OPC UA System Architecture describes the general patterns used in the OPC UA framework. These concepts help understand the high-level concepts used in the OPC UA. The main interest of this thesis, the OPC UA History Gateway, is already partly introduced in the architecture specification.

2.2.1 Client-Server Pattern

The most basic pattern of the OPC UA framework is the already mentioned Client-Server Pattern. In this pattern servers offer services to clients, which consume these services to complete tasks. Services themselves are standalone concepts which contain no calls into each other. [3, Ch. 9, p. 265]

The communication is made possible by strictly defined requests and responses that use service calls defined in the OPC UA Specification (definitions relevant to this thesis are found in section 2.4.2). Clients send requests to servers which return well-formed responses back to clients. The communication is in practice effected using the OPC UA Stack (cf. section 2.3), which, for one, uses TCP, HTTP/SOAP or HTTPS protocols to transfer the information over networks, as already mentioned. [3, Ch. 6] These protocols are widely used in modern Web Services to exchange information and can be used effectively through firewalls.

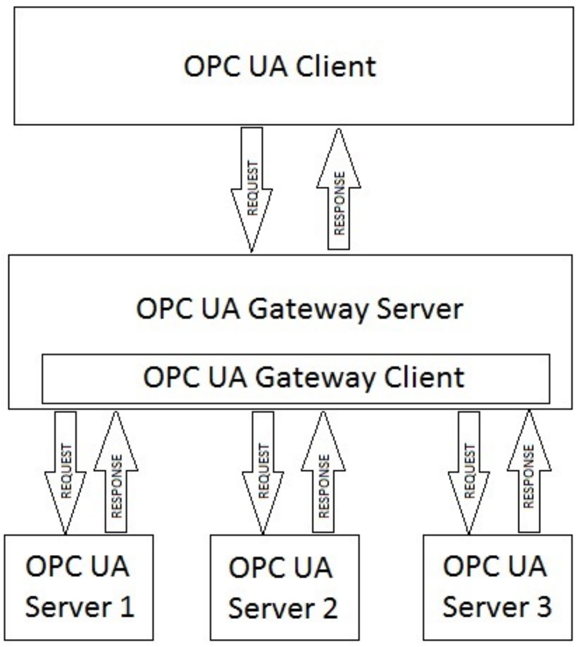


Figure 2: Concept of Aggregating Server (modified from [3])

2.2.2 Aggregating Server

An aggregating server is principally an entity containing both the server and client implementations, and is the most relevant concept for this thesis, describing the basic functionality of the History Gateway (cf. Figure 2).

The aggregating server contains an embedded client, which can send requests to variable number of other servers configured into the Aggregating Server. The data received from these servers is shown in the aggregating server and can be accessed with any client: thus the aggregating server concentrates information. With these properties, the aggregating server is ideal for the History Gateway, as it in effect relays requests from a common client to multiple servers further down in the automation environment.

2.3 OPC UA Application Architecture

Application Architecture clarifies boundaries within and between software. Application Architecture can be defined by the level of Application, SDK and Stack, in descending order of functionality visible to the end user of the software. Figure 3 gives an overview and hierarchy of the concepts.

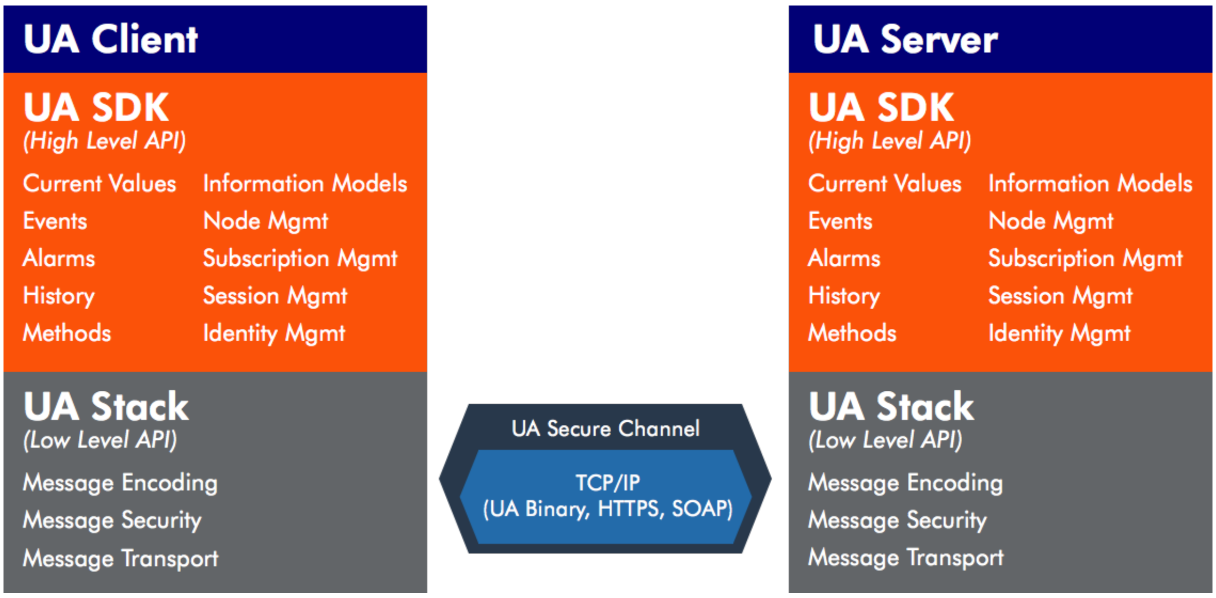


Figure 3: Overview of OPC UA Application, SDK and Stack layers

The Application Layer conforms mostly to the Client-Server Pattern (or an Aggregating Server in the case of the History Gateway), and normally uses the SDK layer as an interface for all the OPC UA specific functionality. All the use case specific functionality is implemented in the Application Layer. In the case of the Server, the Application is in contact with the underlying system or device and uses the OPC UA SDK to model the system in question. The OPC UA acts then as a conduit transporting the modelled information further. In the case of Client, Application is the user's access to the system modelled with the OPC UA.

SDK Layer holds the Server and Client specific APIs (Application Programming Interfaces) in addition to the implementation of the OPC UA concepts described in section 2.4. APIs need to be Client and Server specific, as only Servers process requests and send responses, and vice versa within Clients. SDK eases the Application development considerably by implementing general parts of the somewhat extensive OPC UA framework, along with creating a simplified API to using the Stack. This frees the application developer to concentrate on the features needed in the application (e.g. business or control specific). For example, the Prosys Java SDK was heavily used in creating the History Gateway prototype for this thesis: its use shortened the development time immensely.

OPC UA Stack makes sure that requests from a Client Application are transported to the Server Application and that the responses are sent back to the Client (i.e. that API calls in the SDK layer work). The Stack can be divided into several layers partly shown in Figure 3. The APIs correspond directly to the ones in SDK and act as interfaces to the SDK. Stacks are language specific, and currently there are .NET, Java and ANSI C Stacks available. .NET Stack holds implementations in C# and C++. The implementation of the OPC UA concepts differ somewhat per Stack basis. [10]

2.4 OPC UA Specification

OPC UA is wholly defined by a list of specification documents maintained by the OPC Foundation. Specifications range from basic concepts to the information modelling and network security. [11, 12, 13] Additionally, there are specifications to accommodate the Field Device Integration using OPC UA, a field which is currently divided between few competing standards. [14] Specific solutions to using OPC UA in the Field Device Integration have also been presented. [15] Next subsections go through the OPC UA specifications in more detail, laying out the basics of the Address Space browsing, Event handling, Historical Access and Services. These concepts are essential to understanding what was done in the solution part of the thesis.

2.4.1 Address Space

The Address Space of the OPC UA server defines the structure and data of the modelled system and consists of different types of Nodes. Nodes can be classified by their function and adhere to specific NodeClasses. All NodeClasses hold a specific set of mandatory and optional Attributes, values of which contain the actual data of the Node (and thus the modelled environment). NodeClasses include Object, ObjectType, Variable, VariableType, DataType, ReferenceType, Method and View. NodeClasses and their Attributes are illustrated in Figure 4.

The Address Space is made up of a tree-like structure of Nodes which represent meaningful units in the modelled system. *Object* Nodes are the most general structure in the Address Space and can be used to represent parts of the modelled system. *ObjectType* is abstraction of a concrete object and is thus instanced into an Object.

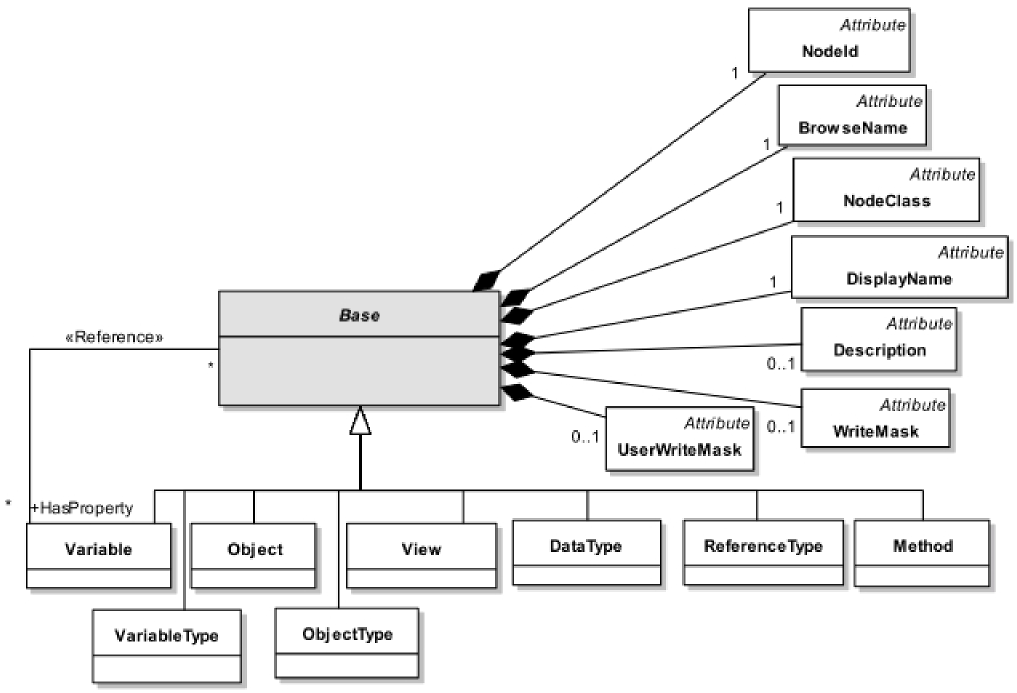


Figure 4: NodeClasses, BaseType and attributes of BaseType [16, p. 84]

The semantics and structure of the Address Space are described using references between Nodes, each reference with a specific *ReferenceType*. Each ReferenceType in part describes a relation between a source and a target Node. *Variables* hold the data values of the system and are instances of *VariableTypes*. Each value of a *Variable* has a specific *DataType*. *Methods* represent callable procedures in the modelled system with defined input and output arguments. *Views* are visible configurations of the Address Space, in a way snapshots, tailored for a specific purpose. This way only the essential information e.g. to a maintenance operator can be filtered from the Address Space using a View.

The most important Attribute found on each Node (as shown in Figure 4) is NodeId, as it uniquely identifies each Node. Address Space is divided into enumerated namespaces having unique URI (Uniform Resource Identifier) values. Namespaces can be thus accessed either using URI or index value. NodeIds themselves also consists of two parts: the first part holds the namespace enumeration and the second part holds String value unique within the namespace. First part can be either the namespace URI or namespace index value. Identifying Nodes from several servers is essential in History Gateway implementation, and is thus a very important aspect of this thesis.

Another important aspect of Address Space is the type hierarchy, which defines the types of Objects, Variables, Data types, Events and References. Types describe the Attributes, Standard Properties and References applicable to the Node in question, making possible the proper use of the Nodes.

The creation of the Address Space in a large environment can be quite complex and time-demanding task, requiring a lot of manual coding. Some tools to help modelling and implementing the Address Space are presented e.g. by Palonen in his Master's Thesis, which devised a strategy to load the Address Space automatically using XML files. [10] To use such automatically created Nodes, however, one needs to resort to manual coding. Laukkanen took the work further in his Master's Thesis by addressing the design of the type instantiation and automatic Java source code generation in the OPC UA framework, alleviating the above problem. [17]

2.4.2 Services

Services handle the actual functionality within OPC UA framework and are used to complete tasks set by users. Services are invoked with a service request and always return a service response. The services reimplemented for the solution of this thesis are covered in this section. Note, however, that Service Sets are not necessarily covered in exhaustive manner.

Generally, all requests contain a RequestHeader, which defines parameters common to all service requests. Examples of these are a time-stamp, authentication token and handle assigned by the client to the request. In the same vein, all responses contain a ResponseHeader, which contains e.g. a time-stamp and result code of the attempted operation. Responses also almost always include diagnostic and result fields.

View Service Set allows browsing the server Address Space for the client. Browsing is done by *Browse* and *BrowseNext* services. Browse returns references of a single Node in the Address Space, from which the next level of the Address Space can be constructed. BrowseNext service is used when browsing a Node with more references than can be handled in a single request. BrowseNext effectively divides the original request into one Browse and multiple BrowseNext requests.

Attribute Service Set handles the actual reading and writing of the Node Attributes. These are done using *Read*, *Write* and *HistoryRead* services. The read service is used to read one or more Attributes of one or more Nodes. Write service conversely allows the client to write one or more Attributes of one or more Nodes. Generally, these are useful when the information within large number of Nodes is accessed. HistoryRead service works in similar fashion with Read service, the difference being that results for each Node span the time interval specified in the request. The historical data structure is defined in more detail in section 2.4.5.

Method Service Set defines the way to use Method Nodes, which are essentially functions with defined input and output arguments. Methods are used with the *Call* service, which returns output values and diagnostics information of the Method called in its CallResponse.

MonitoredItem Service Set is used to keep track of changes in Attributes, Values or Nodes via special MonitoredItems. Figure 5 illustrates the use of the

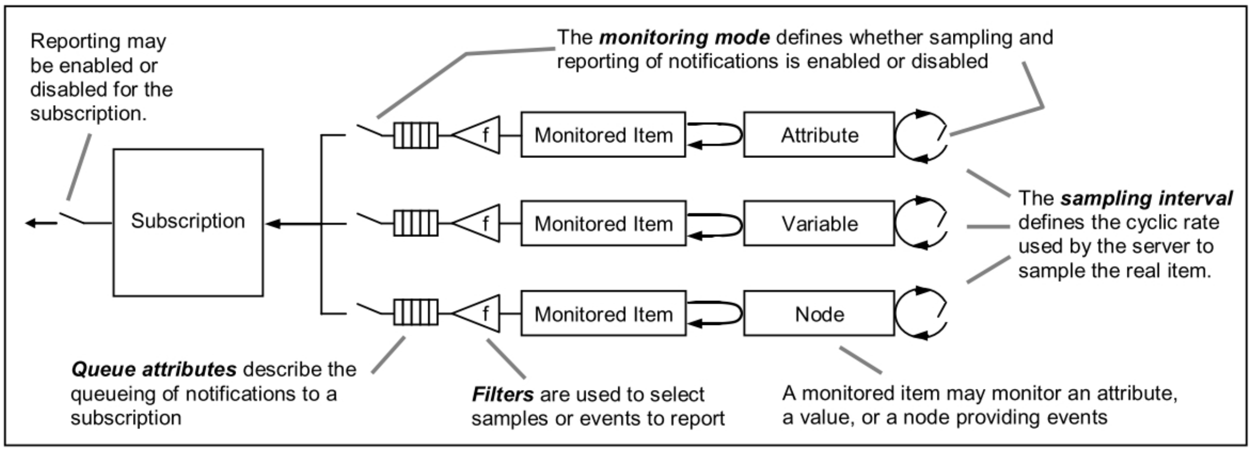

Reporting may be enabled or disabled for the subscription.
The ***monitoring mode*** defines whether sampling and reporting of notifications is enabled or disabled
Subscription
f
Monitored Item
Attribute
Variable
Node
The ***sampling interval*** defines the cyclic rate used by the server to sample the real item.
Queue attributes describe the queueing of notifications to a subscription
Filters are used to select samples or events to report
A monitored item may monitor an attribute, a value, or a node providing events

Figure 5: Subscription with assigned MonitoredItems [18, p. 67]

MonitoredItems. In it each MonitoredItem is assigned to a specific Subscription (defined in the following Service Set). Each MonitoredItem has also properties such as a SamplingInterval, MonitoredMode, Filter and Queue Parameters. These properties define e.g. the frequency by which values are collected and what triggers the change of the value in the Server. MonitoredItem Service Set defines services to *Create*, *Modify* and *Delete MonitoredItems* within a Subscription. MonitoredItems themselves use DataItems and Events to handle the different kind of data in Nodes (cf. sections 2.4.3 and 2.4.4).

Subscription Service Set is used to create Subscriptions for sending Notifications to Clients. As already mentioned, the Subscription holds MonitoredItems assigned by a Client. The Subscription has defined publishing interval, which determines the execution cycle of the Subscription (i.e. checking the value of MonitoredItems). Also, publishing interval determines the default SamplingInterval used for the MonitoredItems. Notifications are, however, only sent as a response to a PublishRequest by the Client (via a *Publish* service). If there are no Notifications to send in a response to the PublishRequest (or within certain number of received PublishRequests), the Subscription sends keep-alive message to the Client instead (which has zero Notifications). The Notification contains the data in question (e.g. value of MonitoredItem) and uses different data structures for DataItems, Events and Subscription status changes.

For an exhaustive list of services and their parameters, reader is suggested to delve into Service Part of the OPC UA Specification. [18]

2.4.3 Data Access

Data Access defines the representation of the data in the OPC UA framework. The data consists mainly of DataItems, a generic description of an entity with specific data type and value changing over time. All DataItems are derived from BaseVariableType.

The standard also defines AnalogItems, DiscreteItems and ArrayItems to serve as containers for different kinds of measurements present in the automation environment. EngineeringUnits are defined for AnalogItems to show the unit of the measurement and to avoid any (costly) confusion regarding it. Figure 6 depicts a typical use of DataItems along with Object in the Address Space. In the Figure pressure information is being held in the Value Attribute of a Pressure Variable of an AnalogItem type. [19]

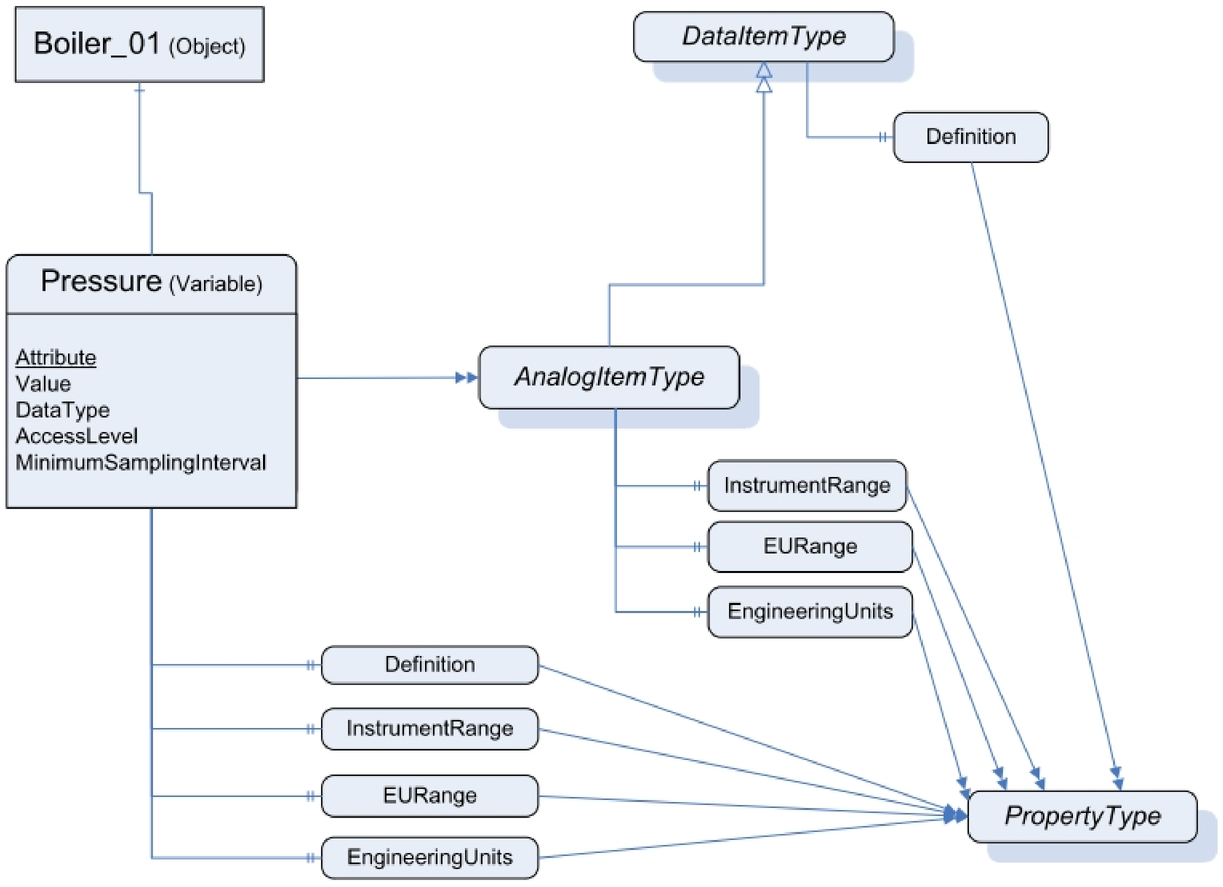


Figure 6: Usage of DataItems within Address Space [19, p. 14]

The Value of DataItems is always the latest measurement received from the device, and is defined by a DataValue type. The type consists of a Value, StatusCode, SourceTimestamp and ServerTimestamp. The Value field holds the data of measurement. A StatusCode states the trustworthiness of the Value and is roughly categorized into a Good, Uncertain and Bad state. A SourceTimestamp is the time in which the value was stored in the device. ServerTimestamp is the time when it was acquired in the server.

DataItems are also used with MonitoredItems to track changes in the value of the variable. What is considered change depends on a DataChangeFilter created separately for each MonitoredItem. Parameters of the DataChangeFilter define the triggering method of the MonitoredItem, i.e. whether a status, value or time-stamp modification (or any combination of above) triggers the sending of the Notification.

The Notification is sent to the Client listening the MonitoredItem via a Subscription. A deadband range of the monitored value is also defined, i.e. the range in which no changes are triggered. The deadband type can be none, absolute or percent. In essence deadband value determines the precision by which the changes are monitored. [18]

2.4.4 Events

Event is a general term for interesting occurrences in the system modelled by the OPC UA framework. [11, Ch. 3.1.10] The purpose of each Event is to report a fulfilment of a predefined rule, which somehow needs to be acknowledged by the user. The data of the Event is contained in its fields. Events can be subscribed to and send Notifications of interesting occurrences via Subscriptions to the Client.

EventFilter is important concept used to select Events in a Subscription, as only the Events matching the EventFilter are sent to the Client. (EventFilters are defined by the Client.) EventFilter defines whereClauses and selectClauses in order to discriminate wanted Events. WhereClauses contain BrowsePath for the Event, defining the identity of the Event. WhereClauses can contain complex criteria for the selection of the Event, and are defined by a ContentFilter type. SelectClauses define wanted fields of the Event, which in most cases is the value of the Event variable. [18, Ch. 7.16.3]

The standard divides Events into Conditions and Alarms (among other types), both of which inherit themselves from the BaseEventType. The distinction between them is that Conditions contain state information, as opposed to every other type.

BaseEventType defines some mandatory fields describing the most important aspects of the Events. These identify a type, source Node, Message and Severity of the Event. The Message is intended to be a localized, human-readable description of the Event. The Severity describes the importance or urgency of the Event in numerical range of 0...1000. The BaseEventType also defines the time of the Event in a similar fashion as was done in the DataValue type, i.e. the time Event was raised and the time Event was received in the server.

Events form somewhat complex hierarchy, part of which is depicted in Figure 7. This complexity, however, makes possible to model the environment with a greater detail without the need to create myriad user- or vendor-specific Event types for the purposes of every application. On the downside, complex hierarchy may confuse someone new to OPC UA framework. [20]

2.4.5 Historical Access

The Historical Access is involved in storing and retrieving data with a temporal dimension. Historical data is divided quite naturally into Event and Data histories, as both have somewhat conceptually different properties: Events track discrete, qualitative changes in the state of the system, while Data Access typically describes the system quantitatively in the form of the process variables.

Historical Data Access involves creating a time series representation of the DataItem data. To this effect, Server wide properties of e.g. a MinimumTimeInterval and Ex-

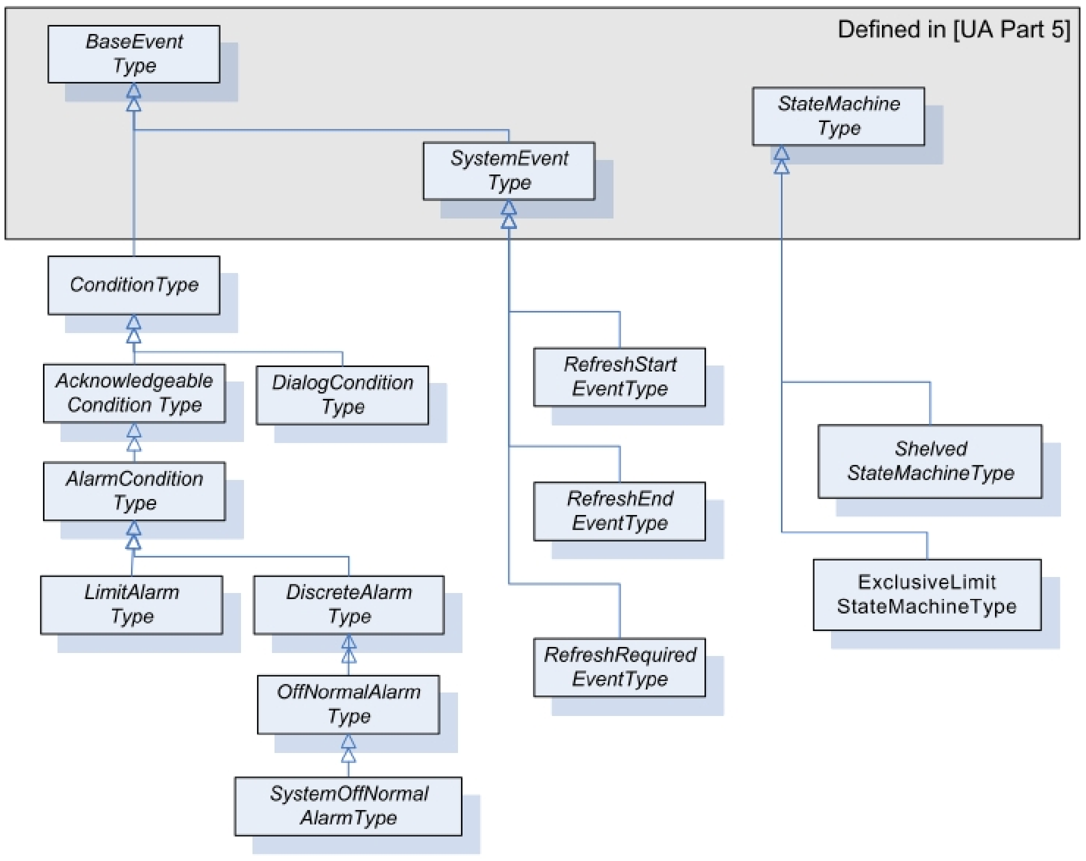


Figure 7: Overview of Event type hierarchy [20, p. 12]

ceptionDeviation are defined, which all historical Nodes adhere to. These define the minimum time difference between data points, and the level of the change needed in order to record a new value into a database. The actual implementation of the persistence is left to the Server (or Client) developer. This thesis concentrates mainly in the issues with storing this type of quantitative data.

Historical Event Access describes the mechanism of accessing and defining Event histories. An Event history Node needs to contain a special HistoryEventFilter, which is of the EventFilter type, describing the fields available in the Server history. This is different from the EventFilters described in the previous section: only the availability of field history is specified, not necessarily the availability of the fields in real-time processing (e.g. via MonitoredItems).

AccessLevel and Historizing Attributes are the main indicators of the historical capabilities of a Node. The AccessLevel indicates whether the Node is readable, writable, history readable or history writable, or any combination of the above. The Historizing Attribute indicates whether the Node is currently collecting History data in the server. [16, p. 27]

HistoryRead Service is used by the Client to access historical data of a Node. Different types of the data are accessed by giving different parameters to this Service. Parameters specify e.g. Read Raw and Read Event functionalities (among others), which allow the reading of the Data and Events, respectively. The Read Raw has a parameter defining a time range, for which DataValues of a number of Nodes are returned by the Server (if available). The Read Event has additionally an EventFilter parameter (used in the same manner as was described in section 2.4.4) specifying the wanted Event. Events matching the time range and the EventFilter are returned as a list by the Server. Other HistoryRead Service functions include reading aggregate values or reading data values at specific time points. In the latter case the server may need to implement an interpolation functionality to offer data exactly at the requested time points.

The specification also defines a *HistoryUpdate Service* to be used in inserting values to an underlying history database. Current generic clients, however, do not use this feature much, and mostly case-specific databases and configurations are built to the server- or client-side.

2.5 Conclusion

OPC UA is a powerful information modelling and managing framework, the possibilities of which are the main interest of this thesis. The concepts of the OPC UA define architecture, communication, security and data modelling capabilities of the system or application: OPC UA is a comprehensive and highly-developed solution to managing the factory floor device space.

The main blocks of the framework are client and server applications, which exchange information between each other via different services. The information model defines an Address Space, which contains Nodes in hierarchical, tree-like layers. Nodes contain references to each other, through which the contents of the Address Space can be browsed. Nodes model the relevant aspects of the underlying system or device. Tools to help this modelling process in a large environment are available. [10, 17]

Applications making use of the benefits of the OPC UA framework are in increase in the industry slowly but surely, and this trend is bound to continue. The main benefit of the framework is the standardized way in which the data is exposed to further processing, already with the semantic information intact. This is especially helpful when integrating the device etc. information into the upper layers of the industrial management systems. [21]

3 Structured Query Language (SQL) Framework

Databases based on Structured Query Language have been around for decades and are still the most widely used solution to storing data in a permanent manner. This is also the case in the automation industry, which motivates their exploration in this thesis.

3.1 Relational Model

The relational model for databases, which was conceived by E. F. Codd in 1970, is now accepted as the standard model for the design of RDBMS (Relational Database Management Systems). [22, p. 1-1] Codd's final book, The Relational Model For Database Management: Version 2, describes theoretical (and at times practical) requirements of a fully realized relational model in the database design, although omitting the actual implementation and syntax completely. [4] This allows the database vendors to create and compete with the underlying implementation, using the relational model as a reference and design document.

3.1.1 Relation and Structure

A relation is defined mathematically using set theory: relation R over sets $\{S_1, S_2, ..., S_n\}$ is a subset of their Cartesian product, i.e.

$$R \subseteq S_1 \times S_2 \times ... \times S_n, \tag{1}$$

with degree of n. [4, p. 2] It is notable that the resulting set of a relation is also a relation, and can be processed further using same operators as was done with the original set. This gives the theoretical basis to the nesting of relational commands.

The data source is addressed as a catalog, and can be thought of as a table (or a collection of tables) for the visualization purposes. The catalog (table) has a number of tuples (rows) and attributes (columns), which contain the actual information of the database. Each column should have a specific meaning, i.e. be a part of some semantically relevant domain. Each row then holds all the pertinent information with regards to the type of the table in question (and forms a single record).

3.1.2 Integrity Constraints

Integrity constraints ensure the incorruptibility and consistency of the information contained in the database. Codd formulated five constraints in his RM/V2 (Relational Model: Version 2) to uphold these properties: Domain integrity, Column integrity, Entity integrity, Referential integrity and User-defined integrity. [4, p. 246] Of these the Entity and Referential integrity are the most important. [4, p. 244]

Entity integrity constraints disallow duplicate rows from the database catalog. This is accomplished by using primary keys which are forced to be unique and not unknown, effectively making all the rows of the table unique. Primary keys can also be formed by combining the values of several columns into a unique identifier. These are called composite keys.

Referential integrity constraints make sure that references between tables stay incorrupt. These references are created, for example, with foreign keys; keys that point to (and actually are) primary keys in some other table. This creates a reference to the other table, effectively linking the two tables. Referential integrity constraints state that no primary keys, which also act as foreign keys, may be removed. [4, p. 23] If this would be allowed, the foreign keys would point to a nonexistent value, and the reference would be corrupted.

Domain integrity constraints ensure that once a domain is specified, all columns part of it must adhere to its properties. Examples of properties might be: the length or range of the value; the default value; or whether the value can be unknown.

Column integrity was introduced to limit the need to create an extensive amount of domains which actually are subsets of other domains. Column integrity is therefore a narrowly defined constraint affixed on a domain.

User-defined integrity constraints allow the database manager to create custom constraints (i.e. to the force business logic on applications).

3.1.3 Transaction

The concept of the transaction defines operations available to changing the state and contents of a database. These are signalled to the database using *Begin transaction* and *Commit transaction* commands. The procedure is two-phased for the option to undo changes in case something unexpected happens. This is called a *Rollback* command and can be initiated any time between begin and commit commands, effectively returning the database to the state before the *Begin transaction* command.

Codd formulated high level commands to access and modify the content of a catalog. These were *Retrieve*, *Insert*, *Update* and *Delete*. Commands are quite self-explanatory: *Retrieve* returns the relation according to the search criterion used; *Insert* inserts information into a new row; *Update* updates existing values in a row; and *Delete* deletes contents of a row.

The commands themselves are put into effect using set theoretic operations of a Select, Project, Join, Product, Union, Intersection, Difference and Divide. [4, Ch. 4] Operations are illustrated in Figure 8.

3.2 Standardized SQL

SQL is currently a standard by ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) organizations, which maintain up-to-date references of the relevant design and implementation aspects of the SQL framework. The actual delivery of these properties depends on the compliance of the providers of RDBMS software (some providers mentioned in section 4).

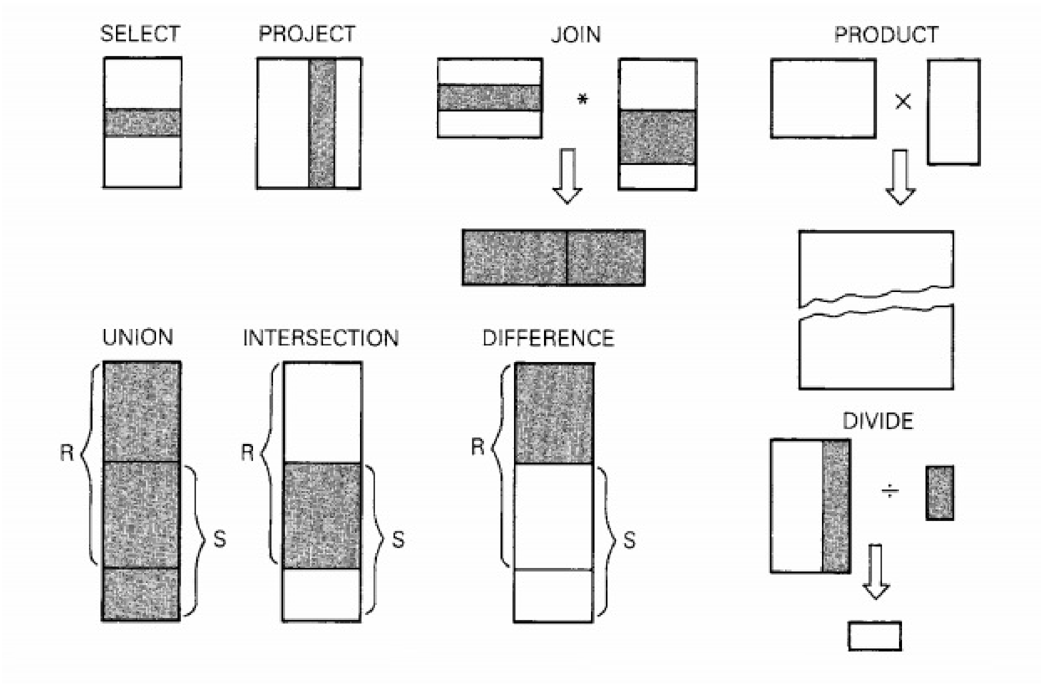


Figure 8: Set theoretic operations used to manipulate relations in relational databases [4, p. 78]

3.2.1 Schema

Schema describes the structure of the database by defining the table names, columns of tables and relations between tables. These are contained in the Information Schema and can be accessed using normal SQL-syntax. The Definition Schema is a hypothetical schema defining possible entities in the Information Schema and in itself. It is defined mostly to be the basis of views in the Information Schema. [5, 6, 23] Schemas are made concrete by tables. Tables consist of columns each with a specific data type, and with one and only one value in each row. [5, Ch. 4.3]

3.2.2 Data Types

SQL Data Types are either *predefined*, *constructed* or *user-defined*. [5, Ch. 4.4] Every data type can contain an unknown value, a keyword for which in most cases is NULL. No comparison can be made between NULL values, even though they can be grouped together using search clauses, for example to find any non-NULL data. The SQL Data Types are of especial importance when the data in object-oriented language is mapped into rows in the relational database. This mapping can be quite complex and its solution is partly addressed in section 5.4.

Predefined types are atomic and can be divided into Numeric, Character string, Binary string, Boolean, Datetime, Interval and XML types. The exact implementation and keywords for each data type vary per vendor, each however fulfilling the functionality of the type. [5, Ch. 4.4]; [6, Ch. 4.2-4.6] In most cases these data types have a straightforward mapping into the data types of the object-oriented programming language, and vice versa.

Constructed types can be atomic or composite. Reference types are the only atomic type, their values pointing to a location containing the value of the referenced type. [6, Ch. 4.9] Composite constructed types consist of Collection, Field and Row types. [6, Ch. 4.10, Ch. 4.8]

User-defined types can be Distinct or Structured types, with a user-specified type name. The values of the Distinct types are based on predefined or collection types. Structured types are fully user-defined and contain attributes, sub- and supertyping which adhere to substitutability: properties more traditionally found in the object-oriented programming languages. [5, Ch. 4.6.4]; [6, Ch. 4.7]

For a more in-depth view of the SQL data types, the reader is suggested to familiarize oneself with ISO/IEC 9075-2:2011 and vendor-specific references. [6]

3.2.3 Statements

The contents of the database are in practice accessed and modified by writing SQL statements, executing them and possibly processing the resulting set. Statements are classified by their function, e.g. whether the statement targets general schema of the database or contents of the table(s). The former statements are in some instances abbreviated as DDL (Data Definition Language) and latter as DML (Data Manipulation Language). [22, 6] SQL standard uses more fine-grained division by the functions of the statements, reporting e.g. Control, Session, Diagnostics and Dynamic statements. [6, Ch. 4.33]

The basic data manipulation is done with statements having structures like:

```
SELECT * FROM [table] WHERE [column] = [value]

INSERT INTO [table] VALUES ([x] [y] [z])

UPDATE [table] SET [columnA] = [valueA] WHERE [columnB] = [valueB]

DELETE FROM [table] WHERE [column] = [value]
```

Commands are quite self-explanatory and were described in section 3.1.3. As the select command only retrieves data from the database without modifying it in any way, it is generally addressed as a query. The schema of a database can be changed by commands:

```
CREATE TABLE

DROP TABLE

ALTER TABLE,
```

where CREATE creates a new table, DROP destroys an existing table, and ALTER modifies the columns of an existing table. Meta-data of the database (e.g. column names and data types of specific table) can be accessed using the Information Schema existing in most database implementations.

Domains and constrains can be formulated with CREATE DOMAIN and CONSTRAINT keywords. Keywords are used to define checks which specify the invalid values of the domain or column. These in practice force business logic on an application, effectively disallowing any prohibited operations.

For a more comprehensive account of the manipulation of database contents (e.g. subqueries, aggregates, SQL functions, triggers and query optimization), the reader is suggested to turn to many reference and text books on the subject. The syntax of these commands also varies more per provider compared to the basic functionality described above.

3.3 SQL Concepts

3.3.1 Index

Index is a hugely important concept in the context of the relational databases, which speeds up retrieving the table contents considerably. This is effected by a dictionary type of implementation, where imposing searches on indexed columns need not to go through the whole table, which in worst case contains hundreds of thousands of rows. [24, p. 11] Instead, index keeps records of the indexed content separately from the table and points to the corresponding rows in the table. This way table rows meeting search criteria are known before accessing the table itself and no traversing of the table is needed.

The index architecture can be *clustered* or *non-clustered*: *Non-clustered* means that the physical order of rows has nothing to do with the indexed order of the rows. This causes no slowdown of the insertion speed as no extra operations regarding the whole structure of the table are needed during the insertion.

Clustered approach, conversely, ensures that the physical order in the disk follows the order of rows in the table. Depending on the ordering of the columns, the clustered approach can speed up the retrieval of the range data significantly. If, for example, the ordering ensures that all the data values of a single Node are ordered beside each other in the table and in the disk, certain range of values reside in a logically ascending or descending order within the disk and the table. Only one continuous block from the table (and disk) is then returned as a resulting set to a this kind of query.

Index types define the data structure of the index and can also affect the performance of the queries depending on the type of data inserted into table. Few most used index types include bitmap, dense, and reversed indices.

Again, for more comprehensive treatment on the principles and design decisions on indexes, the reader is suggested to turn to other resources, which can easily span whole books. [24]

3.3.2 Normalization

Another important design concept with regards to the relational databases is normalization. A database in a normalized form contains the minimal amount of redundancies and dependencies. The objective of the normalization is to make the database free of insert, update and delete anomalies.

The normalization is put into effect by so-called normalized forms developed mainly by Codd in 1970s, although newer formulations exist. [25, 26] In the normalized form data is stored to a number of tables, which are linked together by relations, in an effort to create a logical (in all meanings of the word) separation of the data. This ensures that in the case of inserting data, only appropriate table (containing only one small part of the whole database) needs to be updated, speeding up the insertion process. Another advantage is the easier extension of the database in order to accommodate new types of data often needed in the lifetime of a database, as extensions can be in most cases done simply by adding new tables to the database schema.

However, in the fully normalized form extensive queries can be costly as the data is separated into many tables, the contents of which need to be joined to create a result set for these comprehensive queries. Thus, for performance reasons, the database can be left somewhat denormalized at the discretion of the database administrator.

4 Comparison of Database Management Solutions

4.1 Introduction

Some of the more prominent database products and vendors are: **MySQL**, **SQLite** (open source, for now, rights held by Oracle); **SQL Server** (Microsoft); **Oracle Database** (Oracle); **D2** (IBM); and **postgreSQL** (open source). [24] These solutions in fact implement the SQL (as depicted in ISO/IEC 9075:2011), and can be used in a wide variety of situations from online web stores to industrial process auditing. This means that a somewhat large amount of work is needed to create a fully functional product relevant to each specific case. An other option is to use a provider with an already working analysis and database software that is either designed specifically to a task or is more general in nature. In the latter option the software might still need some integration work to operate fully in the case-specific production environment. All options have their merits and depend largely on the scale and needs of the customer.

There is also a rising interest in a multitude of object-oriented databases (as a counterpoint to the relational approach) stirring in the web service world, mainly for the need to increase the scalability to unprecedented heights. The process industry, on the other hand, has somewhat different needs, as the most important factor is the reliability of the operations and fast-enough insertion times. Thus the rigor of the many SQL products in maintaining the database integrity and the possibility to optimize specific SQL operations are highly appreciated. Mainly for these reasons the object-oriented approaches are not covered in the thesis.

MySQL, SQL Server and PostgreSQL were selected to be put under closer scrutiny for their wide use in the industry and in the products developed by Prosys PMS Ltd. Few complete solutions to combine the database and process handling are also introduced in the conclusion-preceding-section.

Few important aspects to consider are e.g. licensing options, additional tools provided for analysis, replication and clustering options and scalability possibilities. The replication and clustering are the main components of creating HA (High Availability) solutions, i.e. ensuring maximal up-time of the database services. The rigorous quantitative assessment of the criteria mentioned may be difficult, however, due to few independent assessors: much information comes from the providers of the databases themselves, which need to be taken with a grain of salt.

4.2 SQL Server

Microsoft's SQL Server and its various instances are still quite widely in use, according to Gartner and other sources. This section addresses only the latest incarnation, SQL Server 2012.

The mainstream editions of the SQL Server are Enterprise, Standard and Business, all intended for various production environments and needing commercial licenses. The one free edition, SQL Server Express, is heavily downgraded in its performance, and is intended to be used as a testing and learning environment.

(Still, in small and non-performance-critical cases it is perfectly suitable for other uses as well.)

The Standard Edition is suitable for small- to medium-sized cases and contains the features sufficient for a majority of use cases, i.e. core database engine along with reporting and analytic options. The Business Edition contains the features of the Standard Edition enlarged with business intelligence operations, concentrating on the large-scale data management and data analysis. These editions also have basic HA capabilities with 2-node fail-over clustering. The Enterprise Edition is intended for large use cases, coming with the most comprehensive clustering and replication capabilities, in addition to containing all the features of the previous editions.

The licensing options of the Microsoft's SQL Server are based either on the number of cores on the server holding the installation of the database engine, or on the Server+CAL (Client Access License) packages. CAL licenses can be assigned either to a device or a user. This means that either the number of devices or the number of users accessing the SQL Server instance are limited.

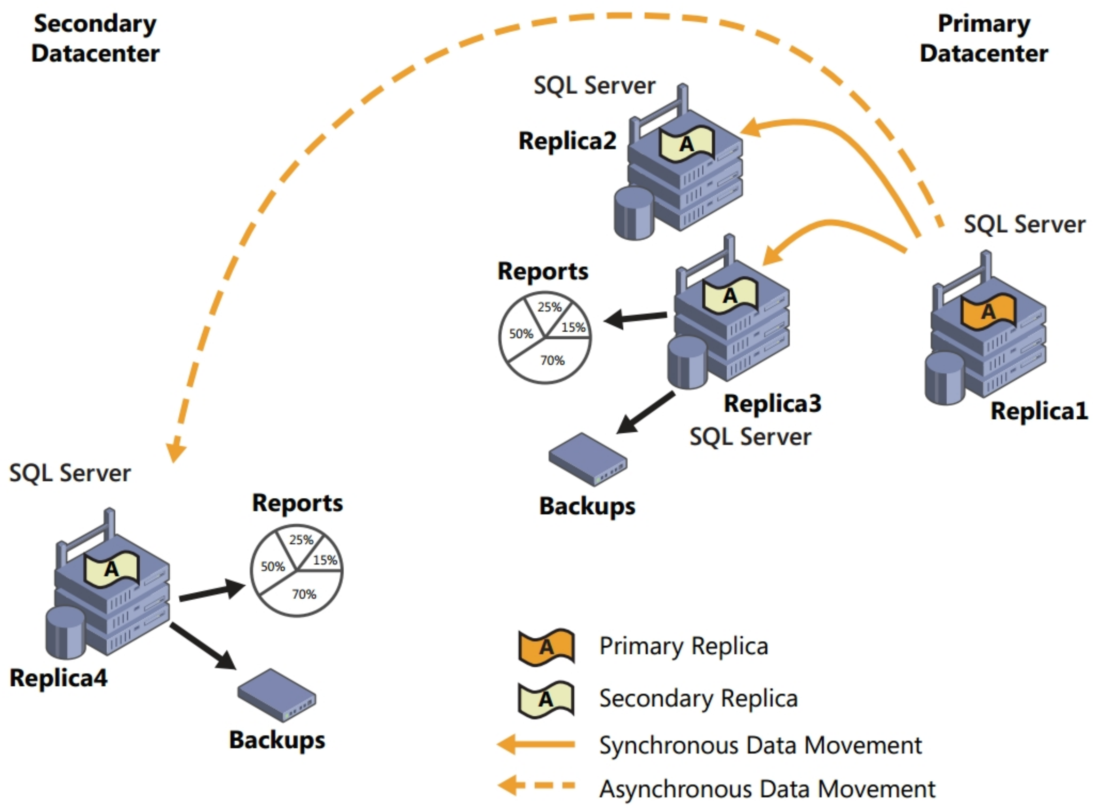


Figure 9: SQL Server enterprise level replication strategy [27, p. 24]

The clustering and replication options in the SQL Server differ somewhat per edition basis. The basic idea is to use active and passive servers, where the state of the active database is mirrored in the passive one. The passive server then becomes active if the currently active server goes down for any reason. The new feature introduced to the SQL Server 2012, AlwaysOn Availability Groups, is intended to answer the HA demands in large enterprise cases. The scheme is illustrated in

Figure 9, in which the replicated SQL Servers in primary datacenter provide the HA capabilities, while the secondary datacenter takes care of recovery in case primary cluster goes down.

All the mainstream editions of the SQL Server come with a somewhat wide tool package called Analysis Services. Analysis Services contain an assortment of data mining tools such as one-layered neural networks, clustering algorithms (Expectation Maximization and K-Means), naive Bayesian algorithms to creating simple Bayesian networks, and time-series prediction (Autoregressive Tree models and Autoregressive Integrating Moving Average models). The problem of these advanced methods lies in that to understand their limitations (and therefore use) one needs extensive experience in the data analysis or machine learning, in which case the tools might be too rigid and black-box oriented to be of any real use (to the professional in the field). [27]

4.3 MySQL

MySQL was originally a fully open-source software, until Oracle bought its developer company MySQL AB. MySQL is now distributed under commercial and free licenses (free with GNU General Public License).

The licences are sold per server basis (each with 1-4 sockets), following a somewhat simpler pattern than the Microsoft case. Commercial editions are known as the MySQL Enterprise Edition and the MySQL Cluster CGE (Carrier Grade Edition). The commercial editions come with a wider technical support and improved feature base, concentrating on the High Availability, scalability and monitoring tools important for maintaining large-scale business operations. The free editions still contain the core database features, including basic replication and partitioning features.

The standard (non-cluster) editions use either InnoDB or MyISAM database engines, of which the InnoDB is the default (and newer) one. The main difference between the engines is the locking principle, which in InnoDB is row-based and in MyISAM is table-based. This means that the concurrency (i.e. the number of simultaneous read and write operations) handling is better in InnoDB, as a smaller amount of the data is in a locked state at any given time. Despite this, InnoDB is still ACID (Atomicity, Consistency, Isolation, Durability) compliant.

The cluster edition is targeted to mission critical operations. Figure 10 illustrates the architecture of the solution. The architecture is designed to survive the loss of any single part of it and still maintain the online status of the business operations.

The cluster database engine differs from the ones used in standard editions, which means that certain operations possible in the standard editions fail in the cluster edition. The cluster database engine is also partly restricted when compared to the standard editions, and some loads are handled more effectively by the standard editions. E.g. maintaining a very large amount of data is not as effective, even though clustering is a good way to scale amount of write operations (as the writes are distributed to multiple disks).

The replication in MySQL is done asynchronously in both the cluster and standard editions, the cluster case being more complex due to a larger amount of de-

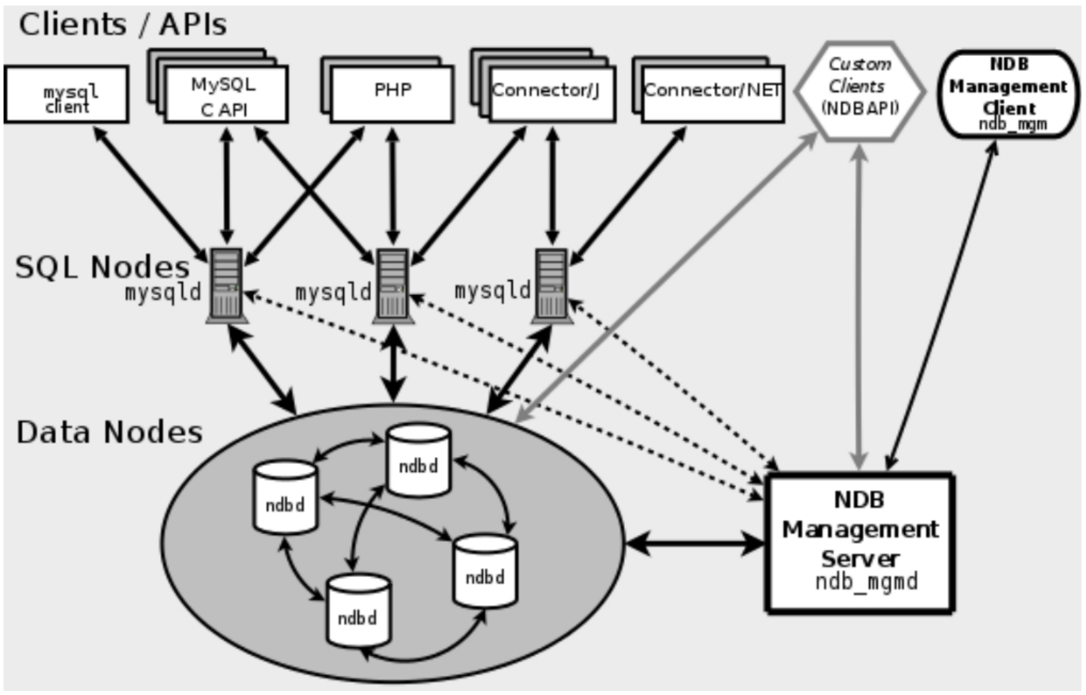


Figure 10: The high-availability architecture of MySQL Cluster implementation [28, p. 2251]

tails in the configurations. Generally, the replication architecture consists of master and slave servers, where the master writes the database operations into a binary log, which is sent to the slave server and from which the slave server updates its state. The asynchronous replication means that the slave(s) need to be connected to the master, but on the other hand the master cannot be completely sure that the commits are being replicated and received. To alleviate this, a semi-synchronous option can be used, in which the master ends the transaction only when the slave acknowledges that the binary log has been received (slowing down the operations somewhat).

The replication can be either statement- or row-based, meaning that either the SQL statement or every changed row per statement is replicated. The row replication ensures that all the changes to the database are replicated in all cases, as long as the log files can be transmitted. The statement based replication produces less log (using significantly less disk space in some cases), but some states of the database cannot be determined solely from the statement parameters.

Additionally, to support a large amount of read operations, *memcached* is often used in the MySQL architecture (available by default in the distribution). It works by introducing RAM-operated (Random Access Memory) buffer, which loads portions of the database to the memory for a read-only access in the client end. This means that if the wanted data is already in the memcached buffer, access is sped up hugely compared to normal queries going through disk I/O. [28]

4.4 PostgreSQL

PostgreSQL is an another open-source database solution, even though there is also a commercial edition (PostgreSQL Plus) available. The latter promise e.g. increased caching capabilities and technical support on its use.

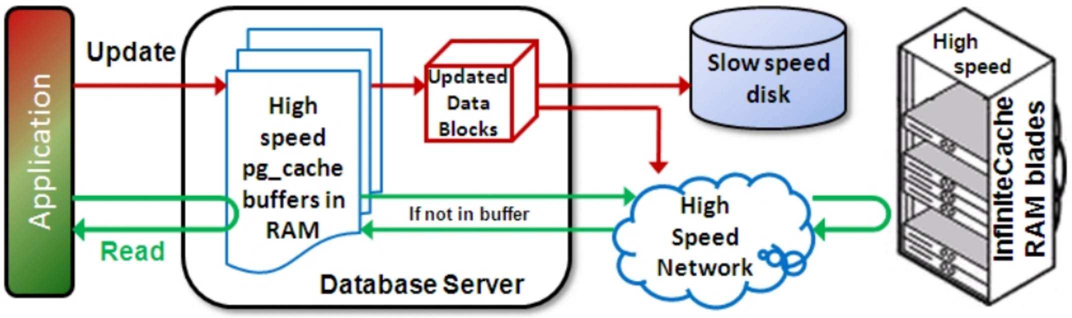


Figure 11: PostgreSQL Plus InfiniteCache architecture [29]

The caching options are somewhat similar to the ones used in MySQL, speeding up the select queries considerably. The commercial edition comes with InfiniteCache, which uses configurable amount of RAM to boost the memory in the cache. The architecture uses primary cache (handled by *pg_pool-II*) and secondary cache (InfiniteCache) in addition to the data compression to make better use of the valuable RAM. The setup is illustrated in Figure 11. The open-source edition uses *memcached* as a caching option in the same manner as the MySQL.

The PostgreSQL replication options contain a single master to a single slave or a single master to multiple slaves architectures. In general the options are more limited than those used in the MySQL. The architecture, however, is somewhat more fault-tolerant, as the database operations are immediately (synchronously) replicated to the passive server (rather than in asynchronous manner as in the standard edition of MySQL). The PostgreSQL also has many open-source additions answering various replication and scaling out needs (such as previously mentioned pg_pool-II).

PostgreSQL has the smallest following of the general database providers addressed in this section, for which reason it has invested considerably to the migration strategies from the larger database providers. This includes various wizard tools for creating a PostgreSQL database out of an Oracle, SQL Server or MySQL one. [30, 31]

4.5 Full-fledged Solutions

To obtain even a crude view of the spectrum of the more refined solutions available to storing and analyzing process data, this section introduces briefly few commercial products to the task. Most of the following systems are intended to handle operations presented in Figure 12, i.e. monitoring and managing the manufacturing process intelligently.

OSIsoft PI System is a large-scale integration solution to managing the industrial environment. The PI System is at the heart of the OSIsoft RtPM (Real-

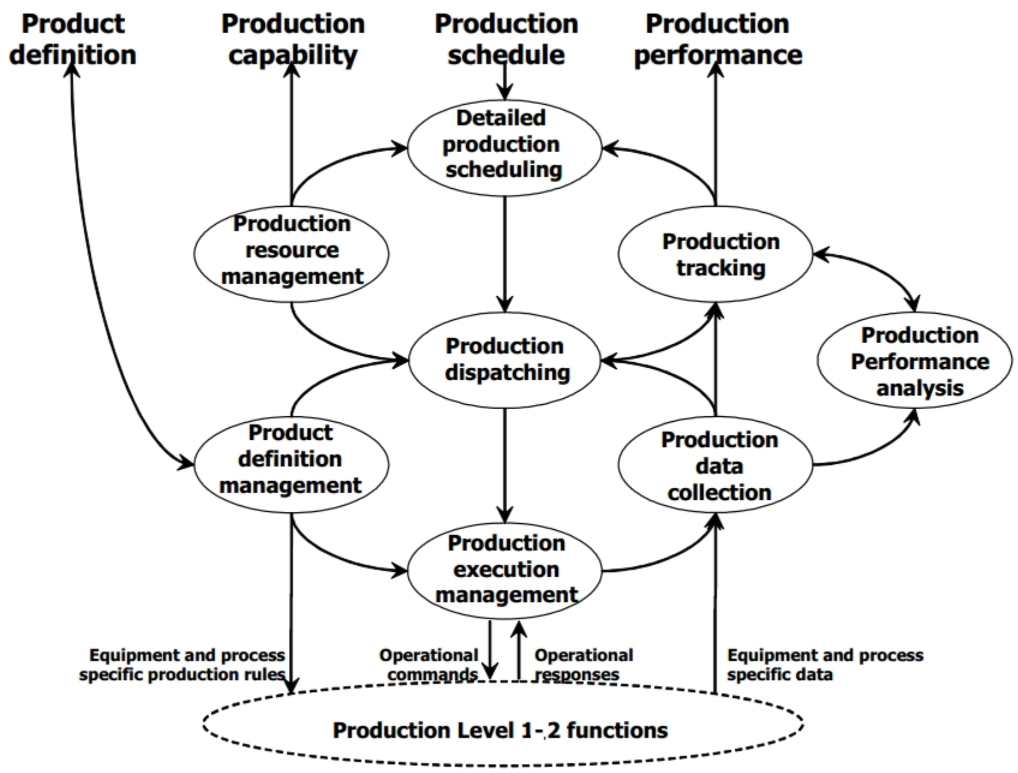

Product definition
Production capability
Production schedule
Production performance
Detailed production scheduling
Production resource management
Production tracking
Production dispatching
Production Performance analysis
Product definition management
Production data collection
Production execution management
Equipment and process specific production rules
Operational commands
Operational responses
Equipment and process specific data
Production Level 1-2 functions

Figure 12: Typical manufacturing operations management functions [1, p. 337]

time Performance Management) system, which conforms to the organizational view presented in Figure 1. The system promises to implement some Level 3 (MES) functions, focusing on the production tracking and data collection (cf. Figure 12). The vertical integration is also taken into account regarding ERP systems such as SAP. PI System also contains basic fail-safe mechanisms and a replication between its interface (client) nodes and servers. [32, 33]

Aspen InfoPlus.21 family is a comprehensive solution to Level 3 functionality, promising to fulfil the operations specified in Figure 12. This is accomplished with a multitude of modules taking care of the integration to the plant floor and analysing the gained data. The post-processing is done e.g. with Real-Time Statistical Process Control Analyzer, which promises to detect the deviation from the output quality in an early stage, along with reducing the waste factor. The tool uses acknowledged SPC (Statistical Process Control) methodology in ascertaining and analysing the process state. All in all, the family contains comprehensive package to process management. [34, 35]

ABB RTDB is the ABB's solution to storing process oriented data in a multitude of fields (from pulp and paper to metal and petrochemical industries). The solution contains its own database implementation named RTDB (Real-time Database), which allegedly is 10 to 100 times faster than the generic solutions presented above. This is needed if a very large amount of tags, e.g. hundreds of thousands, are stored in a real-time fashion (on the scale of one value update per second). To access external data stores, ODBC (Open Database Connectivity) and JDBC (Java Database

Connectivity) drivers are used (cf. section 5). The solution also contains OPC based communication interface for data acquisition, along with extensive analysis and visualization tools implemented using various .NET technologies. The replication and fail-safe mechanisms are not discussed in the software description. [36]

Full-fledged solutions are differentiated somewhat in the comprehensiveness of the offered solution, i.e. if the solution is an all-encompassing MES product or more of a data historian. The technical details of the interfaces to the factory floor level were not addressed in great detail, however, most likely at least the traditional OPC is supported. Similar software that are not explored here come also from Siemens (SIMANTIC Process Historian) and GE (Proficy Historian and Proficy Plant Applications).

As an alternative to the solutions presented above, custom, case-specific modules to data acquisition can be developed, whose out-going messages satisfy the exact protocol used in the company ERP software. This is often the case if full-fledged solutions are not cost-effective, or their comprehensive features are not needed.

4.6 Conclusion

The search for the optimal solution is made somewhat difficult by the lack of hard facts about all the possible options. As the most solutions are products sold and maintained by corporations interested in making profit, the complete performance profiles and technical decisions are not divulged to the public. However, as the objective of the thesis is to find out possible difficulties in *implementing* the database integration, the decision will be limited to the general frameworks.

The main difference between the database solutions is the cost-effectiveness, e.g. MySQL and PostgreSQL are fully functional even with the open-source licenses. Judged by the amount of long-term use, MySQL and SQL Server are proven technologies in the industry. SQL Server also has a largish package of general data analysis tools provided in the Analysis Services. The replication features speak on behalf of the MySQL as the replication can be effectively done using the open-source versions, with components already included in the distribution. The PostgreSQL has multiple solutions for replication, if additional third-party components are used (including compatibility for several database products). However, on the basis of these general notions alone, it is somewhat hard to unambiguously determine the optimal solution: a more refined analysis would have required rigorous performance tests per provider.

As it is, all the general solutions are viable and contain the needed, quite simple functionality (cf. section 6.3) required of the History Gateway. Thus proven viability of the database engine and the available replication and clustering properties were main criteria considered. To that effect, Microsoft SQL Server and MySQL Community Server were used in the History Gateway. Even if PostgreSQL has somewhat good integration capabilities, its use is still quite marginal. Supporting MySQL (Community Server) also ensured cheap deployability of the History Gateway, a cluster case included.

The presented full-fledged solutions pave way for the answer to the research question, *What are the difficulties in integrating industrial process data into SQL database using the OPC UA framework and Java*, by defining the specific needs of the process industry: the ability to save large amounts of near real-time data. Solutions also highlight the need for vertical integration in the systems, as in a full-scale solution the interfaces to ERP (and possibly other MES) systems are vital. An another important aspect is the amount of analysis tools provided by the system and the actual ability to manage the manufacturing process.

5 Database Connectivity and Object Persistence in Java

5.1 Introduction

Accessing a database from any other source than the database management console needs to address the connectivity problem between the application and the database. Some general solutions to connectivity include ADO.NET, ODBC (Open Database Connectivity) and OLE-DB (Object Linking and Embedding, Database), which are middle-ware APIs to accessing a database (or possibly other) content in an analogous way. The technical details of the connection and the capabilities of each of these connection strategies differ somewhat: these differences, however, are not discussed here. There are also some Java-specific ways to solving the connectivity issue. Two of these, full O/RM (Object-Relational Mapping) solution Hibernate and driver-based JDBC (Java Database Connectivity), are examined in this section.

5.2 Object-Relation Mapping Problem

Generally, mapping relational database content into an object-oriented language like Java (and vice versa!) is no straightforward task. [37] There always exists some discrepancy between relations in a database and objects in a programming language, even though in simple cases (e.g. a single object-oriented class with primitive fields), the mapping can be quite trivial to solve. In such case the fields of the primitive data types can be mapped into the columns of a single table, which then contains the data of a single class. This, however, is a very unlikely scenario in an application development.

More generally, the problem can be approached either from the perspective of the relations, which forces the object-oriented application developer to adhere to the data model of the relational database; or vice versa, which forces the relational database to be modelled after the object-oriented classes. In most cases the object-oriented classes contain the business logic in addition to the data. This makes it somewhat natural to use the objects as a starting point for the mapping.

When seen from the viewpoint of the object-oriented classes, the mapping problem includes a representation of class hierarchies (i.e. polymorphism) with some kind of a table structure. General schema solutions are available, however, they produce a large amount of tables and associative tables, which store classes, class hierarchies and attributes within the classes to different tables. With a large class hierarchy, querying such a database becomes very inefficient quickly due to a massive amount of join-operations needed to read the necessary data. Few more specific strategies suitable for different cases exist (cf. section 5.3.2).

In addition to the general class hierarchies, objects hold various relationships or links between each other. These can be divided by their multiplicity into cases of one-to-one, one-to-many and many-to-many.

One-to-one is a straightforward relationship, in which one object is related only to one other object. For example, in the case of a one room flat, one flat holds only one person and one person holds only one flat, i.e. one address.

One-to-many relationship is illustrated by a case of detached house (if we follow the previous example). In it one house holds several people, and each person holds only one flat, or one address.

Many-to-many can be seen as a situation, where one person lives in several addresses, and each house holds several people. This kind of relationship cannot be modelled without the help of an association table, which dissects the many-to-many relation into two one-to-many relations.

Relations between objects can be enriched by association mappings, in which objects have references into each other. These can be illustrated by a parent-child analogy, where the parent has access to the child element and the child element optionally to the parent element. Thus these associations can be unidirectional or bidirectional, and describe whether objects "know" of each other.

More complex situation arises, when the persistent object is of a composite type, e.g. a collection or set. (Some support for these kind of objects may come available with newer versions of SQL, as was briefly described in 3.2.2.) For instance, a case of modelling a collection which contains objects that themselves contain collections might illustrate a case needing quite sophisticated tools in order to allow a straightforward solution.

In general case the problem may surpass mere technical aspects of the mapping: large organizations probably hold quite diverse professionals in the application development and the database management, in which case the problem is also partly interpersonal. The database administrator may have very different ideas on a suitable database architecture than the application developer interested in the object persistence and/or business logic. These types of relations, however, are somewhat out of the scope of this thesis, even if their importance cannot be in any way ignored.

To return to the technical side, Hibernate (by JBoss) promises to solve the mapping issues (some of which were described above) by introducing an abstraction layer between the application and the database. JDBC is a low-level solution mainly solving the connectivity issue, leaving all the mapping to be done by hand by the application developer.

5.3 Hibernate

Hibernate is a somewhat abstract solution for persisting objects in Java, which in an extreme case hides the database completely from the application. Hibernate defines a persistence context, which handles the necessary operations in making the object data persistent in the database.

5.3.1 Architecture

Figure 13 gives an overview of the full Hibernate architecture. The *database* is generally an SQL one (few were introduced in section 4), and holds the data of the persistent objects. *SessionFactory* abstracts the properties of the connection into the database, and is meant to be used from one instance (initialized in the launch of the application). SessionFactory also contains the object-relation mappings to the database used. *Session* is used for doing the actual work regarding the data of the persistent objects and can be obtained from the SessionFactory. Hibernate uses JDBC as a default underlying connection to the database (cf. section 5.4).

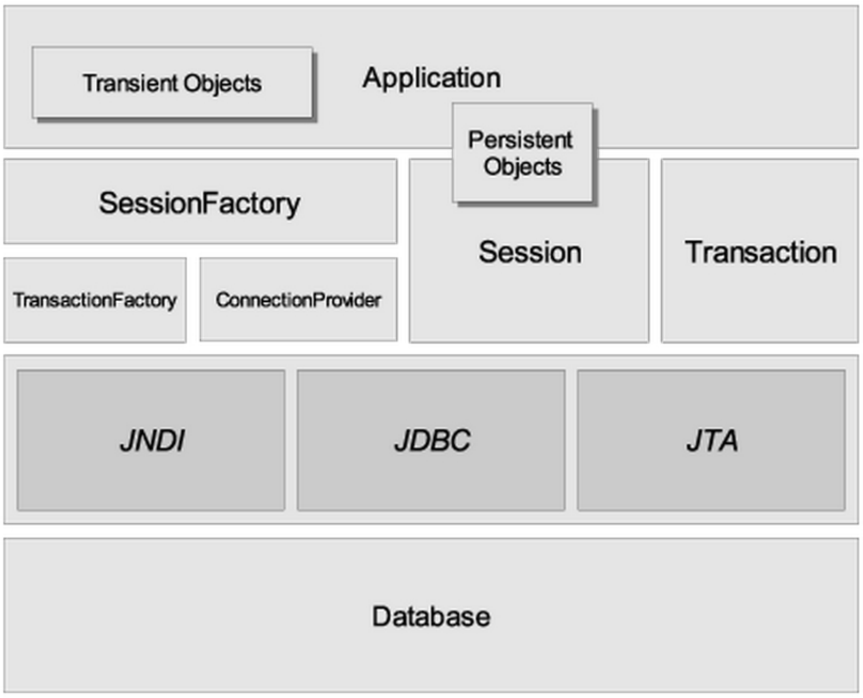


Figure 13: Complete Hibernate persistence architecture [38]

The Java objects in the application are given specific states in order to discern whether the data of an object is up-to-date in relation to the database. These states are *transient*, *persistent* and *detached*, and are modified using the Hibernate loading and saving methods. The transient state means that no persistence is associated with the object. Persistent means that the object is associated with a Session (i.e. used in Save or Load-methods) and is within its scope. The detached state means that the object is out of the scope of any Session, even though it has been associated with one in the past.

Transaction management is a very important aspect of the database access, as on it depends the locking state of the database. Transaction was already defined in section 3.1.3, and is briefly defined as a continuous unit of work done by the database (which can be cancelled while in progress, i.e. within the transaction). In the Hibernate context, transactions are acquired from the Session object, and can be used in managed or unmanaged ways. One option is to use the JTA (Java Transaction API) in Java EE, which provides implementation to handling the transactions (defined e.g. by UserSession interface). If, on the other hand, plain Java SE is used, the Hibernate Transaction API is needed for the handling and demarcation of the transactions.

The environment, in which the Hibernate is used, can also be a somewhat complex entity. Modern corporate IT-systems often contain many layers of abstraction where high level programming frameworks are used (possibly) in abundance. As such, the configuration options in the Hibernate are somewhat complex, and the usage of the Hibernate depends largely on the frameworks in use in the environment. Hibernate integrates into Java environments such as Java EE (which might include Enterprise JavaBeans) by being an implementation of the JPA (Java Persistence API). JPA is a specification of annotations and interfaces, which define how persistence is used in the modern versions of Java.

All the considered details regarding the transaction, session, environment and other scope issues are defined in the configuration, written in XML format. [38]

5.3.2 Hibernate Solutions to Object-Relation Mapping Problem

The Hibernate tackles the mapping of the class hierarchies into relations with a threefold strategy: table per class, table per subclass and table per concrete class hierarchy. The general solution mentioned in the problem description section is omitted altogether by the Hibernate.

Table per class hierarchy means mapping the whole class hierarchy, i.e. all the classes, into one table. This is in practice done by adding the needed amount of columns per class into a table, so that the columns of a single table describe all the properties in all the classes. In the case of the class holding complex objects or lists, the single table structure may become hard to maintain and quite ineffective in the terms of the used space. The table also grows quite quickly even with moderate-sized class hierarchy. Querying such a database is simple, however, as all the properties of the classes are found in a single table. (The problem might lie in discerning what data belongs to a what class, though.)

Table per subclass hierarchy means that every class is mapped to its own table. This configuration is simple to understand, as it generally generates one-to-one mapping between the classes and tables. The properties spanning class hierarchies, i.e. properties found in the superclass, are then modelled with foreign key references. This approach keeps the redundancy of the database in the minimum, as the data is in theory normalized in the process. The querying operations are then somewhat slowed down due to the separation of the data into many tables. (This approach is a toned down version of the general solution, which also maps all the metadata and properties of the class into separate tables.)

Table per concrete class hierarchy is a crossover of the above two strategies, as only the concrete (i.e. instantiated) classes are mapped into a database. However, if the subclasses are not hugely different from each other, the database schema becomes quite redundant by storing the superclass properties over and over again for each concrete class. The advantage of the approach is similar to a moderate denormalization, as the properties of a single class are always found within a single table. The problems of the approach lie in maintaining the data integrity and the large amount of work needed per change in the superclass properties.

The mapping itself between the objects and the database can be done by various techniques, namely with programmatic Java annotations, Java deployment descriptors or XML mapping files. The exact syntactic format depends on the technique used and handles in practice the many-to-many, association, complex object and collection etc. cases. These syntactic details, however important, are not discussed here.

Generally, even with a highly-developed tool such as the Hibernate, the mapping problem is by no means trivial. As can be seen from the large amount of details and possible approaches, there is no single solve-it-all solution applicable to each and every case. This complexity has given the OR/M-solutions somewhat of a bad name. Still, the mapping problem regarding SQL databases is present in many industry cases and its solving cannot be escaped.

5.4 Java Database Connectivity (JDBC)

JDBC (Java Database Connectivity) solves the connectivity issue to the database by a driver-based solution to writing and reading the database content through an API-like Java functionality. The architecture of the JDBC is illustrated in Figure 14. Compared to the previously introduced Hibernate, JDBC solves only the connectivity issue.

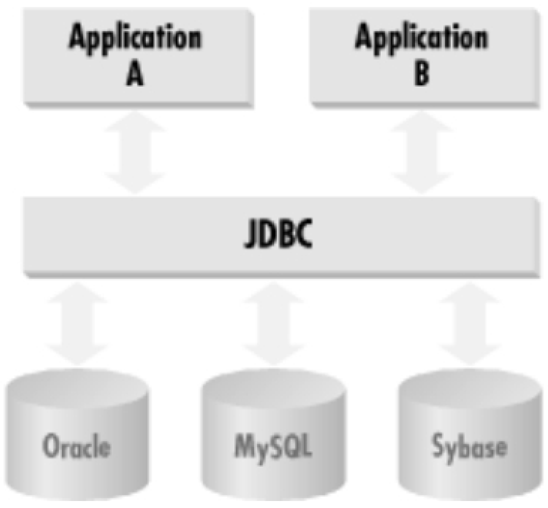


Figure 14: JDBC overall architecture [39, p. 24]

JDBC Drivers can be classified to several types shown in Figure 15. Type 1 drivers are a bridging solution using middle-ware APIs (e.g. ODBC) doing the actual database access. Type 2 drivers contain Java code calling native C or C++ methods provided by the database vendor, i.e. use native API calls. Type 3 drivers are a network solution translating the API calls to database specific calls in the server layer, which in part connects to the database with specific API calls. Type 4 drivers are a pure Java solution communicating with the database directly using database specific protocols. These drivers are mostly supplied and developed specifically by the database provider. Of the drivers presented above, type 4 drivers are most effective, as they contain no intermediate layers in the communication and are completely managed within the Java Virtual Machine. This, at the same time, elicits a platform independence based on Java.

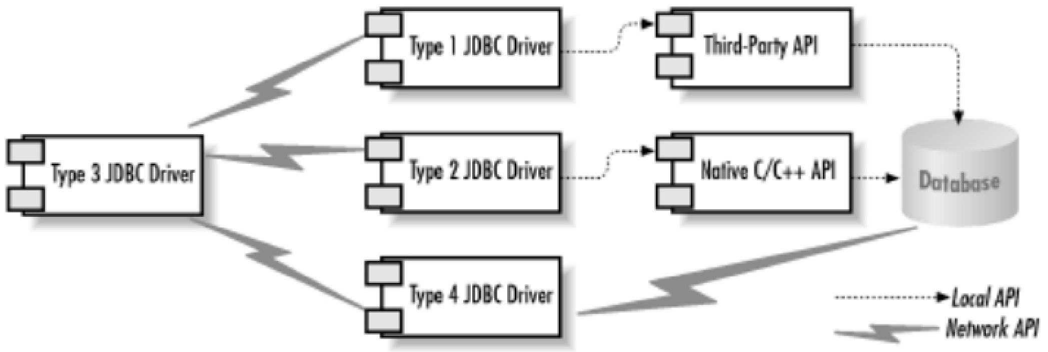


Figure 15: JDBC driver options [39, p. 27]

In practice the access to the database is done using Statement and ResultSet objects in Java. The idea is to write database implementation dependant SQL commands into a Statement object and process the possible results using a ResultSet object. Going through the ResultSet object makes it possible to populate other objects with data. This assumes that the drivers are installed correctly, and that the access to the database is established correctly.

Additionally, all the issues regarding the object-relation-mismatch still remain, leaving the problem-solving regarding it to be coded manually and in case specific manner by the application developer. [39]

5.5 Conclusion

The two technologies here are quite different in their scope and in the solution promised, which makes their evaluation difficult. The Hibernate is a full O/RM-solution solving the mapping problem from the viewpoint of the object-oriented developer. JDBC gives the access only to the database, needing many case-specific solutions to be hand-coded. In the Hibernate the work is done mainly in the configuration, which naturally is case-specific, but can be easily modified to accommodate more complex data schema, if needed. Updating JDBC solution in such case requires most likely modifications to the database saving and loading modules as well, in addition to the considerable modifications to the database itself.

However, since the mapping problem in this thesis is mainly in the level of OPC UA data types (of which some are primitive Java data types, others derivatives of primitive types) versus SQL data types, a straightforward solution with the JDBC was opted for. The data consisted mainly of MonitoredDataItems, which are composed of a data value, status code, time-stamp and ID used in the server (cf. section 2.4.3). Thus there was no need to accommodate rich class objects in the database schema, and really no need for a full-fledged O/RM in the form of the Hibernate. Also, as the main interest in the process industry is keeping the database up-to-date, i.e. high enough insertion speed, the overhead of mapping objects to tables through XML could be a disadvantage, even though probably a small one. Still, the considerable time increase in the development process and the possible optimization of a new technology spoke negatively of the Hibernate (in the scope of this thesis).

This section also answered the general part of the question *What are the difficulties in integrating industrial process data into an SQL database using the OPC UA framework and Java*, i.e. what are the general difficulties using SQL databases from the viewpoint of the Java (cf. section 5.2). The answer is also valid in most modern (object-oriented) programming environments. Additionally, the section presented Java-specific solutions (via Hibernate) to the defined problem (cf. section 5.3.2).

6 OPC UA History Gateway

6.1 Introduction

The integration of the information from the factory floor to the upper levels of the industrial organization can be done using the OPC UA framework and OPC UA History Gateway. In practice this is done by mediating information between the different OPC UA clients and servers. The History Gateway server can contain the contents of many underlying servers, acting as an aggregate server, from which data of many normal OPC UA servers can be managed simultaneously. [3, p. 268] Within the OPC UA framework, the History Gateway resides between the Application and SDK Layer, as a generic OPC UA client can use the History Gateway as its interface into a multitude of OPC UA servers.

More generally, the History Gateway resides between Levels 2 and 3 in the industrial scheme (cf. Figure 1). In theory, however, many of the functionalities needed in the Level 3 (cf. Figure 12) could be implemented in the History Gateway or directly on top of it. Still, in the context of this thesis, the History Gateway is seen more as a data acquisition and storing system than a fine-grained analysis and management system.

6.2 Use Cases

One of the main problems of creating an advanced PCMS (Process Control Monitoring System), which monitors a whole industrial plant and calculates performance metrics, is the lack of sophisticated trend data. [40] The main use case of the History Gateway is to create such trends out of the process data, as early as possible in the organizational hierarchy. Any advanced analysis software needs data to function, making the acquisition part the first problem to be solved.

In a more simple use case, the History Gateway can be utilized in managing multiple OPC UA servers using a single interface (as it is possible to configure several servers into the Gateway). This makes the plant management more effective e.g. for control engineers, and in fact brings about the device space integration. This is also a starting point to reducing time delays in the vertical control loops up to the ERP level. [41]

Both above goals assume that the plant is using OPC UA (i.e. there is device information available in an OPC UA server), which currently is not the case in many environments. This problem, however, can be alleviated for most parts by wrapping traditional OPC servers (which are a somewhat de facto standard in the industry) into OPC UA ones. The remaining option of devices not supporting any communication through OPC is exceedingly rare.

6.3 Requirements

To answer the above use cases, requirements for the History Gateway were the ability to process and handle essential Service Requests to the underlying servers and return valid results. Other main functionality was the application's ability to access some SQL database directly, in order to store and retrieve different kinds of History data.

More specifically, the most essential OPC UA services to be implemented were the View Service Set, Attribute Service Set, Method Service Set, MonitoredItem Service Set and Subscription Service Set (refer to section 2.4.2 for the descriptions of the above Sets).

The requirements for the database integration included solving all the practical connectivity and mapping issues raised in section 5 in effective manner. Connectivity was required to the SQL implementations evaluated and selected in section 4.

Software design requirements were the ability to use Prosys Java SDK, in addition to allowing the efficient relaying of service calls and database integration. Requirements for the SQL Information Schema were small to none data redundancy accompanied with fast query and update capabilities (as was applicable). The Information Schema also needed to incorporate OPC UA DataValue data.

6.4 Solution Foundation

As mentioned in the introduction, the Prosys OPC UA Java SDK was taken as a starting point for the implementation of the History Gateway. The SDK already contained sample implementations of the OPC UA client and server software. These were fully utilized in the development of the History Gateway, and provided along with the rest of the SDK a framework to which implement the novel features. This way the OPC UA framework could be easily utilized in the History Gateway solution. Still, OPC UA Java Stack functions were used extensively to create the needed features of the Gateway (refer to section 2.3 for details of Stack functionality).

6.5 Solution Overview

One main question, *What are the difficulties in relaying service calls in the OPC UA framework*, is defined by keeping track of the communication between one OPC UA Gateway server and multiple underlying OPC UA servers. (This general schema was already illustrated in Figure 2.) Connections between servers are abstracted with namespace maps, where each namespace in the History Gateway is mapped into a namespace of another OPC UA server down the chain. The connection itself is made by an OPC UA client instance, which further uses the OPC UA Stack to communicate with the wanted server instance.

Figure 16 illustrates the main problems which the History Gateway solves in the initialization of the software. These problems are mainly about the access and unification of multiple UA Server namespaces. First, the History Gateway reads a list of UA Server endpoints, whose Address Spaces it incorporates. An equal number of UA clients are created for accessing the UA servers. Address Spaces are not intended to be stored fully, as the relaying is done per namespace basis. Namespaces per server are stored in the database and a unified namespace for the History Gateway Address Space is created. If any new UA server is encountered, a

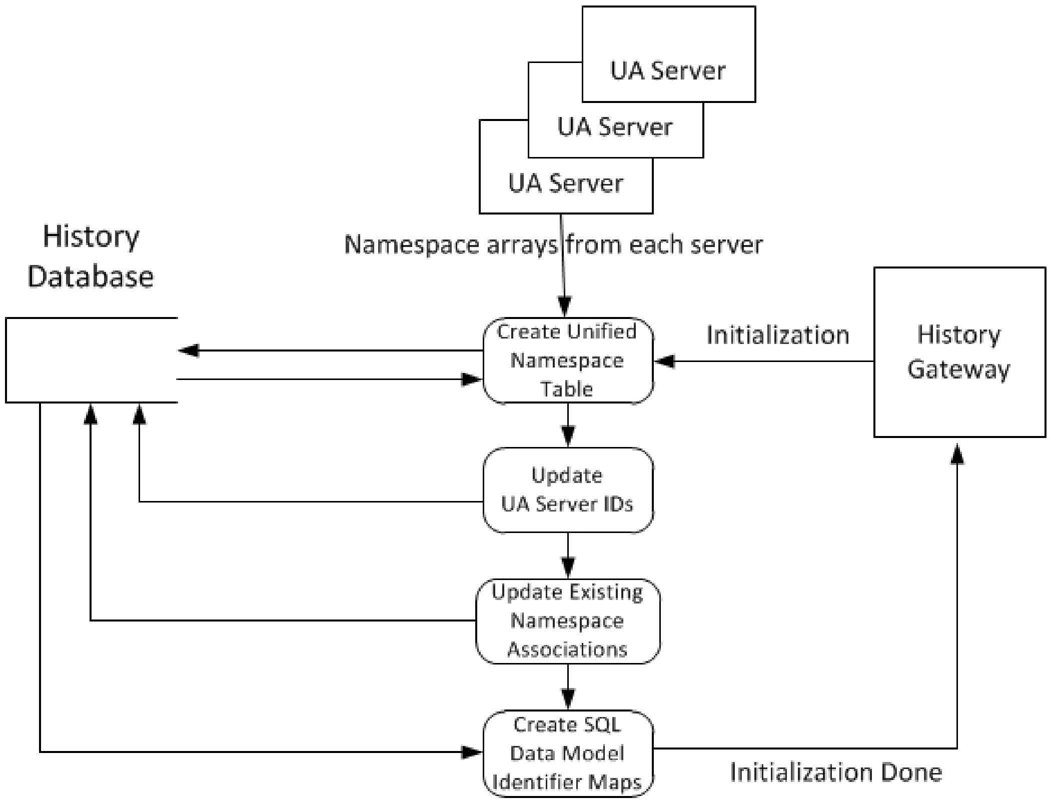

UA Server
UA Server
UA Server
History Database
Namespace arrays from each server
Create Unified Namespace Table
Initialization
History Gateway
Update UA Server IDs
Update Existing Namespace Associations
Create SQL Data Model Identifier Maps
Initialization Done

Figure 16: Unification of Several UA Server namespaces

fixed server ID is created for it in the database. Any changes in the already read namespaces need to be taken into account, possibly changing the stored associations between servers and namespaces in the database. Finally, HashMaps of the essential database mappings are created (the SQL Data Model is introduced in section 6.7.2). At this point the History Gateway accepts connections from UA clients, allowing the Browsing of its unified Address Space.

Another important aspect of the solution is the structure which makes possible to complete all the above procedures. Figure 17 presents the software structure of the History Gateway in the form of an UML diagram (only key fields and methods included). The diagram illustrates the answer to the research question *What is a suitable software structure to accommodate the relaying of service calls and SQL database integration.* The design adapted as much as possible to the structures present in the SDK to allow seamless use of the SDK in the History Gateway. This allowed, for example, to use the already implemented structures to handling the communication through OPC UA Stack, and to keep the solution in the level of the Service Requests and Responses.

The idea of the design is to allow GatewayUaClient class to keep track of all the underlying UA servers (accessed via generic UA clients), and save the values of the MonitoredDataItems into the database (via a HistoryDatabase instance). GatewayUaServer class holds the implementations of the UA Service Sets, through which the History Gateway is used by any generic UA client. These Services then

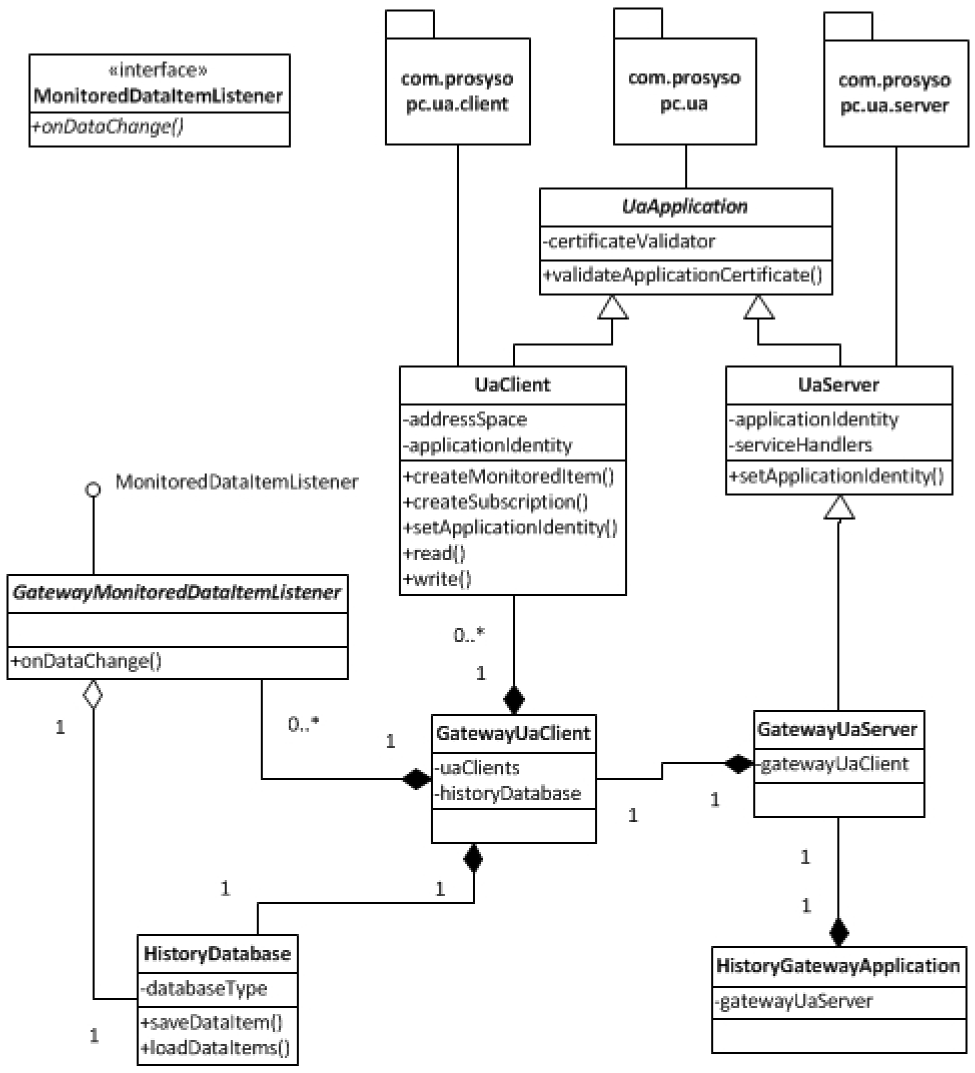


Figure 17: Architecture of History Gateway

use the GatewayUaClient class in order to relay all the service calls. This makes the History Gateway server and client implementations deeply coupled, which, however, is quite inevitable in the gateway scheme.

6.6 Relaying of OPC UA Service Calls

The software design per Service Set and ServiceHandler (which do the actual communication in the OPC UA framework) is shown in Figure 18, and follows the OPC UA specification quite closely. (Some interfaces were combined into a single ServiceHandler in order to keep their amount in check.) Each Service Set (and Service) had to be overridden in order to create the functionality needed for relaying the Service calls (e.g. divide the original Service Request into smaller requests, one for each of the underlying servers). Practical decisions regarding the most important Service Sets are briefly described next, and answer the first part of the research question

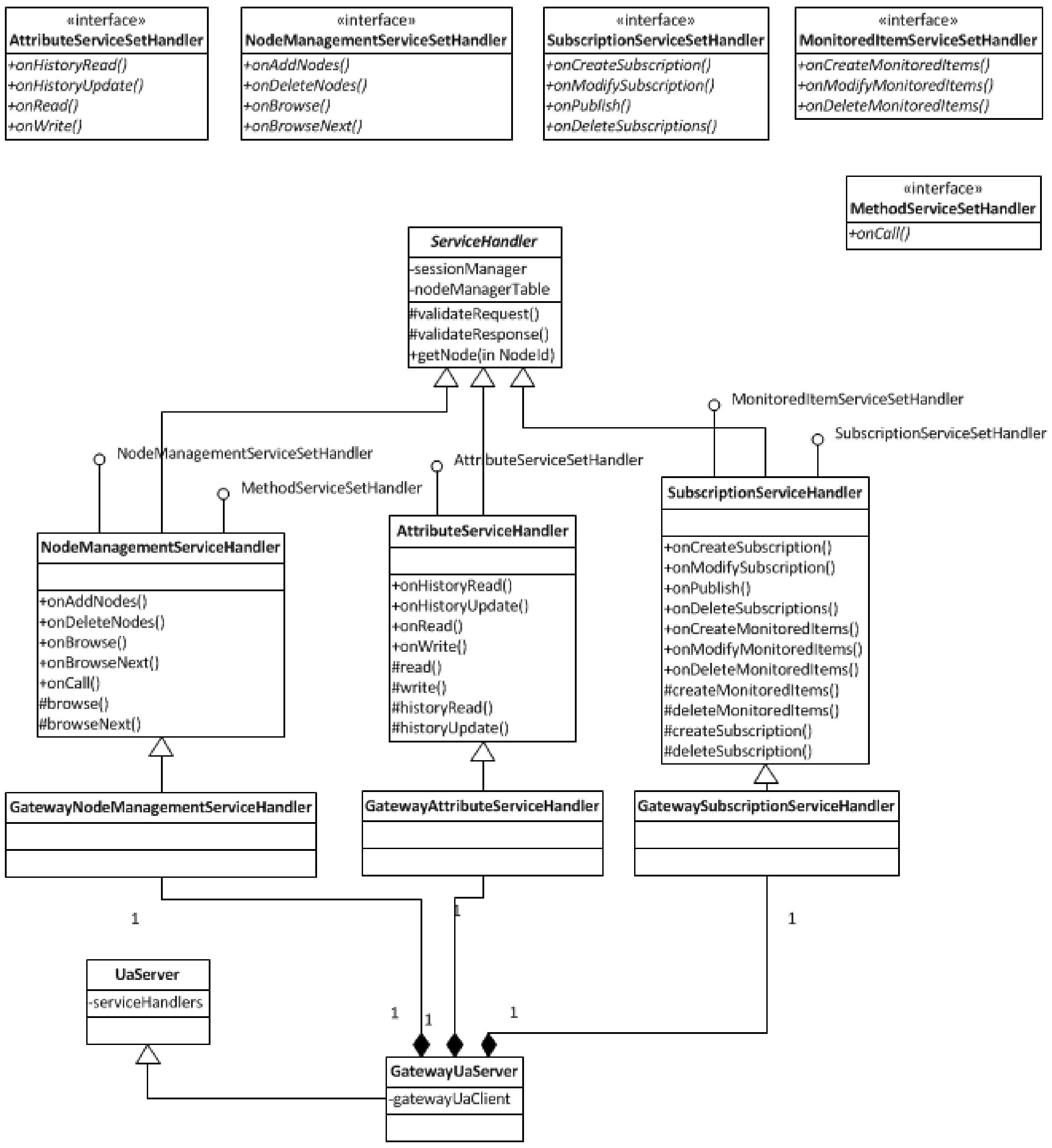


Figure 18: Overview of the implementation of OPC UA Services

How can the relaying of service calls and SQL database integration be implemented in Java.

View Service Set was combined into a NodeManagement Service Set in the Prosys Java SDK interfaces, as seen in Figure 18. Browse and BrowseNext are thus implemented in the NodeManagementServiceHandler. Browse and BrowseNext functionalities were straightforwardly relayed into a single server, as one branch of the Address Space contains only one server. The main level contained placeholders for the root of the each underlying server, from which their contents could be browsed further. The Services first modified NodeIds to match the underlying server (before sending the Service Request), and then modified the returned References to match Gateway namespaces (in the Service Response).

Attribute Service Set holds Read, Write and HistoryRead Services, which are implemented in the AttributeServiceHandler. Each Service needed to be modified so that NodeIds conformed to the namespace used in the History Gateway, however, at the same time showing the data from the correct underlying source. To this effect, every NodeId is modified into a correct one (in underlying server context) before the Request is sent and then modified back to show a correct NodeId (in the Gateway context) in the Response. Also, the HistoryRead Service was set to use the database implementation in showing the trend data of the wanted Nodes. The database implementation returned an array of DataValues for each Node from which the time series were constructed. Service also needed to implement Service Request splitting and merging in order to handle a request containing Nodes from multiple underlying servers.

Method Service Set was also combined into the NodeManagementServiceHandler as it consisted only of a single Call Service, which was straightforwardly relayed to an underlying server. One method call can point only to a one server.

MonitoredItem Service Set was combined into the SubscriptionServiceHandler as all the MonitoredItems need to be assigned into a Subscription in any case. Create and Delete MonitoredItem Services were implemented to take into account the mapping between the shown MonitoredItem (in the Gateway server) and the MonitoredItem being monitored in the underlying server (via the Gateway client). These mappings needed to keep track of the SubscriptionIds and MonitoredItemIds in both realisations along with the correct namespaces of the NodeIds. Because of myriad details concerning the MonitoredItems (i.e. DataItems and Events, their filters, server and client Subscriptions etc.), this functionality proved to be the hardest to implement. Appendix A contains an example of the Create MonitoredItem Service implementation (the server part).

Subscription Service Set was implemented in the SubscriptionServiceHandler by adding Create and Delete Subscription services into it. The implementation created Subscriptions in the Gateway server and client in order to make the link from the client to underlying servers possible. This meant that items in a Gateway Server Subscription were possibly mapped to the several Gateway client Subscriptions, mapping of which was done in the Create MonitoredItem Service implementation. This is also illustrated in Appendix A.

6.7 SQL Integration

This section answers the second part of the question *How can the relaying of service calls and SQL database integration be implemented in Java*. First of all, the database is accessed using a single database class within the History Gateway. The instance of the class is accessed by a MonitoredDataItemListener to save the data values of the MonitoredItems into the database when they change (cf. Figure 17). To this effect, a DataChangeFilter is used to detect changes in the DataValue (as was described in section 2.4.3). As pointed earlier, DataValues are loaded from the database using the HistoryRead service.

6.7.1 JDBC-based Interface and Database Configuration

As section 4 argued, the main connectivity was opted for the MySQL and SQL Server. For these RDBMS, the JDBC drivers were used to access the SQL databases. These were of the type 4 (cf. section 5.4) and were provided by the database vendors. The interface was implemented by writing SQL commands to JDBC objects and executing them. To support both the MySQL and SQL Server, two sets of SQL commands were needed to be written, one for each database type. Examples of the storing and retrieving commands are found in Appendix B (cf. section 6.7.2 for the used SQL Data Model).

A clustered index was used in the data value table as most of the time user is interested in values within a certain time interval. Thus the values are stored in order and the range retrieval is sped up. The Node and source time-stamp were used as indices, which effectively inserts data values of each unique Node in continuous lumps ordered by their time signature.

It is notable, though, that the index configuration should depend on the most common operation on the database. If the contents of the database are not read often, or reads are not time critical, non-clustered index would probably suit the database better. In that case the saving of the new values, i.e. insertion, is not hindered (this was discussed in section 3.3.1).

6.7.2 SQL Data Model

Figure 19 depicts the ER (Entity-Relationship) diagram of the SQL database data model. The schema is separated into tables identifying the underlying UA servers, unified namespace, NodeIds and all the stored values of the NodeIds. The tables are normalized to ensure the uniqueness of the servers and NodeIds along with enforcing data integrity: primary keys with foreign key references between the tables are in use.

In the scheme above, servers are identified by their Server Application URI and Server Endpoint URI, of which the Server Application URI should be unique identifier for a single piece of software [42]. The Server Application URI combined with the Server Endpoint URI can identify servers even in the case of multiple instances of the same software within a single machine. ServerId is a database specific surrogate key simplifying references to the servers. ServerId is in turn mapped to the varying configuration of UA clients in the History Gateway.

Namespace URIs are stored in the schema to discern changes in the configuration of the underlying servers. The ServerIds and namespace URIs are mapped through association table, which is updated at the software start-up. This makes the database independent of the changes in the unified namespace index values. Also, new namespaces URIs can be added to underlying servers without database corruption.

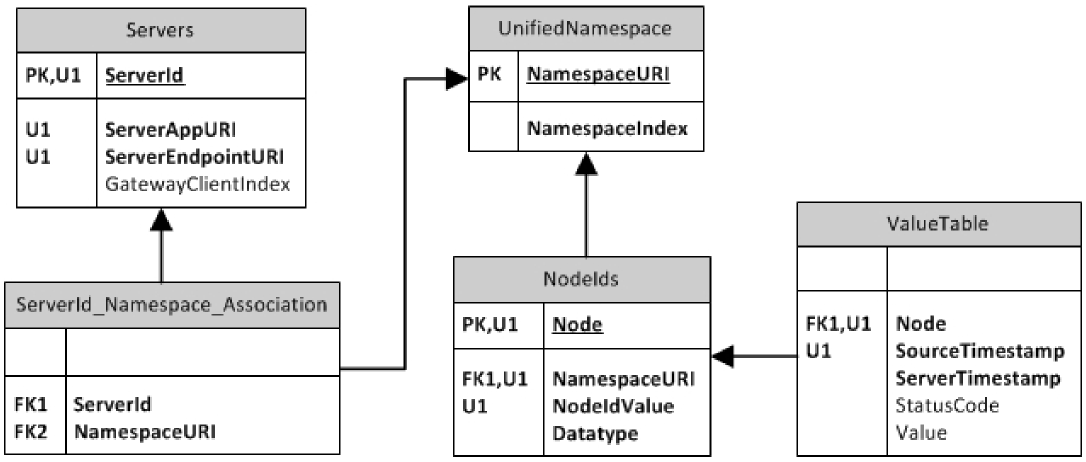


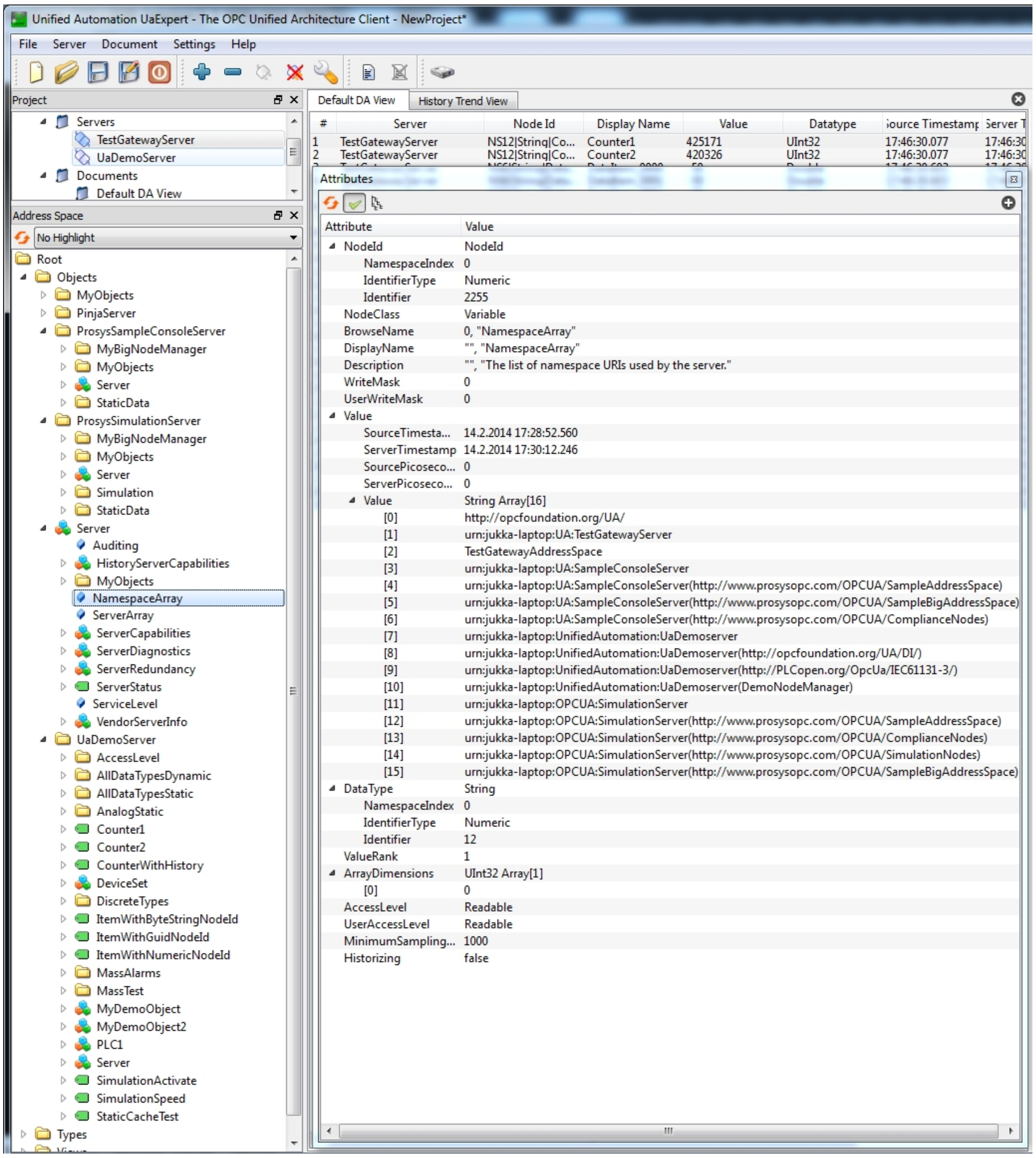
Figure 19: Entity-Relationship diagram of History Gateway database data model

NodeIds are identified with the namespace URI and NodeId String value, whose combination is unique. The NodeIds are also given a surrogate key in the database to separate the database key from the possible changes in the NodeId identifier value. Lastly, the ValueTable holds the data values of all the Nodes, which are indexed by the surrogate Node key and the source time-stamp. A HashMap implementation is used in identifying the Nodes from the ValueTable. This gets rid of the slow joining of the UnifiedNamespace and NodeId tables each time the data values are accessed.

The UA data types were handled through Java type casting (some types specific to the OPC UA) with all the values residing as the text data type in the SQL table. The casting implementation is presented in Appendix C. It is worth noting, however, that in the case of the more complex data types (e.g. arrays, XML and any composite types in general), the value tables need to be data type specific, resulting in as many value tables as there are data types. Event histories were also left out from the data model in the prototype phase, which simplified the data model considerably. Section 6.8 addresses features transcending the prototype status of History Gateway, including the amount of the supported data types.

All in all, the schema answers the research question *What is a suitable SQL database structure to storing data modelled in the OPC UA framework*. Even if the answer covers only a small part of the OPC UA data model, it is applicable to a large part of the process data, namely the continuous process variable data. The used solution combined a normalized database with the exclusion of the join queries, which was possible due to the somewhat simple nature of the (object) data model. The few attributes identifying the Nodes can be easily held in HashMaps during the application runtime.

Figures 20 and 21 show the functionality of the History Gateway when accessed by a generic OPC UA client (by Unified Automation). Figures show the unification of several OPC UA Address Spaces along with the acquired trend data spanning several OPC UA servers.



Servers
- TestGatewayServer
- UaDemoServer

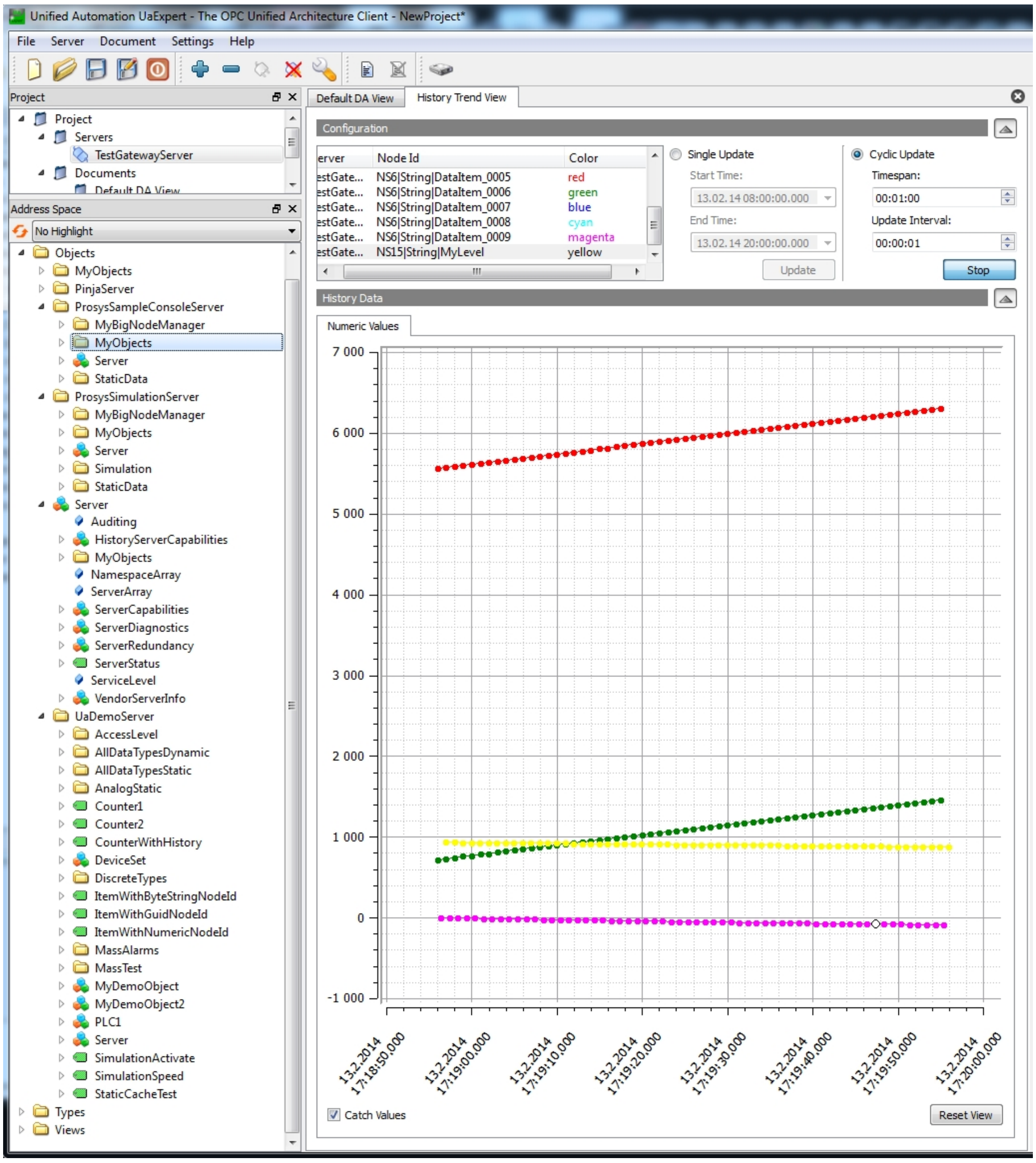
Documents
- Default DA View

Root
- Objects
  - MyObjects
  - PinjaServer
  - ProsysSampleConsoleServer
    - MyBigNodeManager
    - MyObjects
    - Server
    - StaticData
  - ProsysSimulationServer
    - MyBigNodeManager
    - MyObjects
    - Server
    - Simulation
    - StaticData
  - Server
    - Auditing
    - HistoryServerCapabilities
    - MyObjects
    - NamespaceArray
    - ServerArray
    - ServerCapabilities
    - ServerDiagnostics
    - ServerRedundancy
    - ServerStatus
    - ServiceLevel
    - VendorServerInfo
  - UaDemoServer
    - AccessLevel
    - AllDataTypesDynamic
    - AllDataTypesStatic
    - AnalogStatic
    - Counter1
    - Counter2
    - CounterWithHistory
    - DeviceSet
    - DiscreteTypes
    - ItemWithByteStringNodeId
    - ItemWithGuidNodeId
    - ItemWithNumericNodeId
    - MassAlarms
    - MassTest
    - MyDemoObject
    - MyDemoObject2
    - PLC1
    - Server
    - SimulationActivate
    - SimulationSpeed
    - StaticCacheTest
- Types

#	Server	Node Id	Display Name	Value	Datatype	Source Timestamp	Server T
1	TestGatewayServer	NS12|String|Co...	Counter1	425171	UInt32	17:46:30.077	17:46:30
2	TestGatewayServer	NS12|String|Co...	Counter2	420326	UInt32	17:46:30.077	17:46:30

Attribute	Value
NodeId	NodeId
NamespaceIndex	0
IdentifierType	Numeric
Identifier	2255
NodeClass	Variable
BrowseName	0, "NamespaceArray"
DisplayName	"", "NamespaceArray"
Description	"", "The list of namespace URIs used by the server."
WriteMask	0
UserWriteMask	0
Value	
SourceTimesta...	14.2.2014 17:28:52.560
ServerTimestamp	14.2.2014 17:30:12.246
SourcePicoseco...	0
ServerPicoseco...	0
Value	String Array[16]
[0]	http://opcfoundation.org/UA/
[1]	urn:jukka-laptop:UA:TestGatewayServer
[2]	TestGatewayAddressSpace
[3]	urn:jukka-laptop:UA:SampleConsoleServer
[4]	urn:jukka-laptop:UA:SampleConsoleServer(http://www.prosysopc.com/OPCUA/SampleAddressSpace)
[5]	urn:jukka-laptop:UA:SampleConsoleServer(http://www.prosysopc.com/OPCUA/SampleBigAddressSpace)
[6]	urn:jukka-laptop:UA:SampleConsoleServer(http://www.prosysopc.com/OPCUA/ComplianceNodes)
[7]	urn:jukka-laptop:UnifiedAutomation:UaDemoserver
[8]	urn:jukka-laptop:UnifiedAutomation:UaDemoserver(http://opcfoundation.org/UA/DI/)
[9]	urn:jukka-laptop:UnifiedAutomation:UaDemoserver(http://PLCopen.org/OpcUa/IEC61131-3/)
[10]	urn:jukka-laptop:UnifiedAutomation:UaDemoserver(DemoNodeManager)
[11]	urn:jukka-laptop:OPCUA:SimulationServer
[12]	urn:jukka-laptop:OPCUA:SimulationServer(http://www.prosysopc.com/OPCUA/SampleAddressSpace)
[13]	urn:jukka-laptop:OPCUA:SimulationServer(http://www.prosysopc.com/OPCUA/ComplianceNodes)
[14]	urn:jukka-laptop:OPCUA:SimulationServer(http://www.prosysopc.com/OPCUA/SimulationNodes)
[15]	urn:jukka-laptop:OPCUA:SimulationServer(http://www.prosysopc.com/OPCUA/SampleBigAddressSpace)
DataType	String
NamespaceIndex	0
IdentifierType	Numeric
Identifier	12
ValueRank	1
ArrayDimensions	UInt32 Array[1]
[0]	0
AccessLevel	Readable
UserAccessLevel	Readable
MinimumSampling...	1000
Historizing	false

Figure 20: The unified History Gateway Address Space and Namespace Table as seen by a generic UA client



erver	Node Id	Color
estGate...	NS6|String|DataItem_0005	red
estGate...	NS6|String|DataItem_0006	green
estGate...	NS6|String|DataItem_0007	blue
estGate...	NS6|String|DataItem_0008	cyan
estGate...	NS6|String|DataItem_0009	magenta
estGate...	NS15|String|MyLevel	yellow

Single Update
Start Time: 13.02.14 08:00:00.000
End Time: 13.02.14 20:00:00.000

[✓ Cyclic Update]
Timespan: 00:01:00
Update Interval: 00:00:01

[✓ Catch Values]

Figure 21: History Gateway trend data from multiple UA servers as seen by a generic UA client

6.8 Future Possibilities and Features

The future development of the software would need to take into account the supported data types, Event histories, pre-configuration of the MonitoredItems, among other things. A sketch of the new features and their update schedule is illustrated in Table 1. Supported types could include e.g. arrays, matrices (of varying dimensions) and range types. The configuration of the servers would be ideally done by XML files, needing the implementation of the XML serialization.

Feature \ Version	Prototype	Alpha	Beta	Full Product
Core UA Services	x	x	x	x
SQL Integration	x	x	x	x
Basic Datatype Handling	x	x	x	x
Complex Datatype Handling (Array, XML etc.)		x	x	x
Event Type Handling + EventHistory		x	x	x
Configure NodeIds For Saving Using CSV / XML File			x	
Full HistoryRead Service Functionality			x	x
HistoryUpdate Service			x	x
Configuration Utility with Graphical User Interface				x
Offline Address Space Functionality				x
Scale-up Solution				x

Table 1: History Gateway current (prototype) and future feature list

Also, the OPC UA history read functions (such as reading minimum, maximum or average aggregate values; reading at interpolated time points etc.) would need to be implemented. Currently only the ReadRaw-parameter is used, which returns the data values at exact time points for which the data is available.

HistoryUpdate Service would also be a natural place to handle the database storing operations. This would possibly involve developing an UA client supporting the Service. In any case the NodeIds of the saved data items should be configurable somehow. An another option would be to develop a small configuration utility, which would accept lists of NodeIds to be saved in various formats (XML, CSV). Configuration could be even done in an interactive graphical fashion depending on which Nodes are read from the underlying servers.

Other improvements might include increasing the integration capabilities of the software, for example the ability to access (or wrap) traditional OPC DA servers. However, wrapping is already possible using the Unified Automation UaGateway, which could be employed per OPC DA server basis. Vertical integration to the upper layers of hierarchy is quite ERP (or MES) specific and general solution to that the problem is somewhat hard to devise.

Another large-scale update would be to allow the browsing of the History Gateway Address Space (and Node) contents without online access to the underlying

UA servers. This could be done, for example, with the help of the latest history database values. A representation of the underlying Address Spaces would need to be created in addition to the logic of handling the differences in online/offline-functionality. The user should also be clearly notified that the data is only from the database, which might not be evident simply by browsing the Address Space. The offline functionality would essentially replicate the Address Space of the underlying server(s), along with the Type hierarchy. This would be a somewhat large undertaking.

To scale the solution up, a hierarchical deployment of the History Gateways would be possible. In this scheme, a small number of Gateways would be deployed in the lowest level, whose Address Space would then be read by a single Gateway in the upper tier. In this way, a very large number of Nodes residing in multiple UA servers could be read by dividing the Nodes to different Gateways. The database functionality would then be divided into multiple servers reducing the load per server. The uppermost Gateway could still show the full Address Space containing all the Nodes, with some delay. The uppermost Gateway would also need to turn off its database functionality for the improvements to ensue, turning it into a normal OPC UA Gateway.

7 Conclusions

The vertical (as well as horizontal) integration of the information and its analysis are key elements in the modern industrial process management. This thesis has illustrated the viability of the OPC UA History Gateway with inherent database functionality in making that integration reality.

The History Gateway can be seen as an upgrade to low-level data acquisition software, closing to the functionality of the full-fledged solutions presented in section 4.5. However, in its current form, the History Gateway is really standing between Levels 2 and 3 (cf. Figures 1 and 12) of the industrial schema: the History Gateway can publish and store structured data spanning most of the plant floor, easing the integration of the process variables and devices to the MES level and beyond.

To accomplish the modelling, storing and integration of the process data, several strategies need to be used. The modelling part has been addressed extensively in the theses by Palonen and Laukkanen, this thesis addresses the rest. [10, 17] Multitude of challenges presented by above tasks is reflected in the research questions, answers to which are found throughout the thesis. In sum, the answers defined difficulties in the process data integration and persistence in conjunction with the OPC UA, SQL and Java.

The first research question, *What are the difficulties in relaying service calls in the OPC UA framework*, is answered by the need to form links between the History Gateway server and several OPC UA servers. The difficulties lie in unifying multiple Address Spaces into a single one in the History Gateway server. Also, a custom OPC UA client is needed to handle the communication to the underlying servers. All the OPC UA service requests in the History Gateway need to go through the custom OPC UA client with an access to the underlying servers. Also, the obtained results need to be modified to conform to the unified Address Space of the History Gateway. Difficulties and solutions per Service Set are addressed in detail in section 6.6.

The second research question, *What are the difficulties in integrating industrial process data into an SQL database using the OPC UA framework and Java*, is answered by the objective of storing a large amount of real-time process data into a database. Thus possible difficulties lie in the efficiency of the database solution. Also, the general difficulties in using an object-oriented programming language to store data in an SQL database is known as an OR/M (Object-Relation Mapping) problem and is addressed in detail in section 5.2. The problem is about storing the objects, object associations and class hierarchies into a relational database table structure. The OPC UA related problems are about representing the OPC UA data in the SQL data model.

The answer to the third research question, *What is a suitable software structure to accommodate the relaying of service calls and SQL database integration*, is illustrated mainly by Figures 17 and 18. The software structure makes possible to solve the issues raised by the first research question, mainly by coupling the OPC UA client and server implementations. The design relies on the Prosys Java SDK to take the full advantage of the OPC UA communication implemented in the Stack level along with the already implemented service framework.

The answer to the fourth research question, *What is a suitable SQL database structure to storing data modelled in the OPC UA framework*, is illustrated by Figure 19. The database data model allows the storing of the OPC UA DataValues (cf. section 2.4.3) to the database, along with the namespaces which define the Address Space of the History Gateway. This way the design also addresses the problems raised by the second research question.

The fifth and final research question, *How can the relaying of service calls and SQL database integration be implemented in Java*, is answered most thoroughly in the Appendices and in section 6. In sum, the solution utilizes the database data model along with some HashMap implementations to solve the Address Space unification problem (by unifying the namespaces of the underlying UA servers). The implementation issues regarding the namespace unification are illustrated in Figure 16. The storing part is shown to be solved using the MySQL and SQL Server, along with JDBC drivers and objects within Java. The solution also answers the OR/M problem regarding the OPC UA DataValue data. All in all, the developed prototype as a whole illustrates an answer to the final research question.

In conclusion, this thesis has shown OPC UA to be the future of interoperability in the industrial automation world, a fact which is hopefully evident in the OPC UA History Gateway solution.

References

[1] V. L. Trevathan, *A Guide to the Automation Body of Knowledge*, 2nd ed. International Society of Automation, 2006.

[2] American National Standards Institute and International Society of Automation, *ANSI/ISA-95.00.03-2005: Enterprise-Control System Integration, Part 3: Models of Manufacturing Operations Management*, 2005.

[3] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*. Berlin: Springer Berlin Heidelberg, 2009. [Online]. Available: http://www.springerlink.com/index/10.1007/978-3-540-68899-0

[4] E. F. Codd, *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[5] International Organization for Standardization and International Electrotechnical Commission, "ISO/IEC 9075-1:2011 Part 1: Framework (SQL/Framework)," 2011.

[6] ——, "ISO/IEC 9075-2:2011 Part 2: Foundation (SQL/Foundation)," 2011.

[7] P. Ceruzzi, *A History of Modern Computing*, 2nd ed. The MIT Press, 2003.

[8] V. Tan and M.-J. Yi, "Flexibility and Interoperability in Automation Systems by Means of Service Oriented Architecture," in *Advanced Intelligent Computing Theories and Applications. With Aspects of Artificial Intelligence*, ser. Lecture Notes in Computer Science, D.-S. Huang, X. Zhang, C. Reyes Garcia, and L. Zhang, Eds. Springer Berlin Heidelberg, 2010, vol. 6216, pp. 554–563. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14932-0_69

[9] H. R. H. Renjie, L. F. L. Feng, and P. D. P. Dongbo, "Research on OPC UA security," pp. 1439–1444, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5514836

[10] O. Palonen, "Object-oriented implementation of OPC UA information models in Java," Master's Thesis, Aalto University, School of Eletrical Engineering, Espoo, 2010.

[11] OPC Foundation, "OPC Unified Architecture Specification, Part 1: Overview and Concepts, Release 1.02," 2012. [Online]. Available: http://opcfoundation.org/UA/Part1

[12] ——, "OPC Unified Architecture Specification, Part 2: Security Model, Release 1.01," 2009. [Online]. Available: http://opcfoundation.org/UA/Part2

[13] ——, "OPC Unified Architecture Specification, Part 5: Information Model, Release 1.02," 2012. [Online]. Available: http://opcfoundation.org/UA/Part5

[14] D. Grossmann, K. Bender, and B. Danzer, “OPC UA based Field Device Integration,” in *SICE Annual Conference, 2008*, 2008, pp. 933–938.

[15] V. Tan, D.-S. Yoo, and M.-J. Yi, “Device Integration Approach to OPC UA-Based Process Automation Systems with FDT/DTM and EDDL,” in *Emerging Intelligent Computing Technology and Applications. With Aspects of Artificial Intelligence*, D.-S. Huang, K.-H. Jo, H.-H. Lee, H.-J. Kang, and V. Bevilacqua, Eds. Springer Berlin Heidelberg, 2009, pp. 1001–1012.

[16] OPC Foundation, “OPC Unified Architecture Specification, Part 3: Address Space Model, Release 1.02,” Jul. 2012. [Online]. Available: http://opcfoundation.org/UA/Part3

[17] E. Laukkanen, “Java source code generation from OPC UA information models,” Master’s Thesis, Aalto University, School of Eletrical Engineering, Espoo, 2013.

[18] OPC Foundation, “OPC Unified Architecture Specification, Part 4: Services, Release 1.02,” 2012. [Online]. Available: http://opcfoundation.org/UA/Part4

[19] ——, “OPC Unified Architecture Specification, Part 8: Data Access, Release 1.02,” 2012. [Online]. Available: http://opcfoundation.org/UA/Part8

[20] ——, “OPC Unified Architecture Specification, Part 9: Alarms and Conditions, Release 1.02,” 2012. [Online]. Available: http://opcfoundation.org/UA/Part9

[21] V. Tan and M.-J. Yi, “OPC UA Based Information Modeling for Distributed Industrial Systems,” in *Advanced Intelligent Computing Theories and Applications*, ser. Lecture Notes in Computer Science, D.-S. Huang, Z. Zhao, V. Bevilacqua, and J. Figueroa, Eds. Springer Berlin Heidelberg, 2010, vol. 6215, pp. 531–539. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14922-1_66

[22] Oracle, *Oracle Database: SQL Reference*, 10th ed., 2003. [Online]. Available: https://www.stanford.edu/dept/itss/docs/oracle/10g/server.101/b10759.pdf

[23] International Organization for Standardization and International Electrotechnical Commission, “ISO/IEC 9075-11:2011 Part 11: Information and Definition Schemas (SQL/Schemata),” 2011.

[24] A. Hovi, *SQL-Opas*, 1st ed. Jyväskylä: Docendo Finland Oy, 2004.

[25] E. F. Codd, “A Relational Model of Data for Large Shared Data Banks,” *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970. [Online]. Available: http://doi.acm.org/10.1145/362384.362685

[26] ——, “Further Normalization of the Data Base Relational Model,” *IBM Research Report, San Jose, California*, vol. RJ909, 1971.

[27] R. Mistry and S. Misner, *Introducing Microsoft SQL Server 2012*, H. Anne, M. Devon, and D. Carol, Eds. Microsoft Press, 2012.

[28] Oracle, "MySQL 5.6 Reference Manual," 2014. [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/

[29] EnterpriseDB, "Breaking the Scalability Barrier with Infinite Cache," 2009. [Online]. Available: www.enterprisedb.com

[30] ——, "A Comparison of PostgreSQL 9.0 and MySQL 5.5," 2011. [Online]. Available: www.enterprisedb.com

[31] ——, "Comparing MySQL and Postgres 9.0 Replication," 2010. [Online]. Available: www.enterprisedb.com

[32] OSIsoft Inc., "ISA 95 and the RtPM Platform," Tech. Rep., 2005. [Online]. Available: http://www.osisoft.com/resources/white_papers/White_Papers.aspx

[33] ——, "High Availability PI," 2006. [Online]. Available: http://www.osisoft.com/resources/white_papers/White_Papers.aspx

[34] Aspentech, "aspenONE MES," 2013. [Online]. Available: http://aspentech.com/products/aspenONE-MES/

[35] H. Meyr, J. Rohde, M. Wagner, and U. Wetterauer, "Architecture of Selected APS," in *Supply Chain Management and Advanced Planning*, H. Stadtler and C. Kilger, Eds. Springer Berlin Heidelberg, 2005, pp. 341–353. [Online]. Available: http://dx.doi.org/10.1007/3-540-24814-5_19

[36] ABB, "Industrial IT Process Data Management," 2006. [Online]. Available: http://www.abb.fi/industries/db0003db001873/c12570c30026b341c12570c10042dba8.aspx

[37] S. W. Palmer, "The Object-Relational Impedance Mismatch," 2013. [Online]. Available: http://www.agiledata.org/essays/impedanceMismatch.html

[38] JBoss, "Hibernate Reference Documentation," 2014. [Online]. Available: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/

[39] G. Reese, *Database Programming with JDBC and Java*, 2nd ed. O'Reilly & Associates, Inc., 2000.

[40] L. Desborough and R. Miller, "Increasing customer value of industrial control performance monitoring - Honeywell's experience," in *Preprint of Chemical Process Control, CPC-6*, Tucson, Arizona, 2002, pp. 153–186.

[41] T. Gerber, A. Theorin, and C. Johnsson, "Towards a seamless integration between process modeling descriptions at business and production levels: work in progress," *Journal of Intelligent Manufacturing*, pp. 1–11, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10845-013-0754-x

[42] R. Armstrong and P. Hunkar, “The OPC UA Security Model For Administrators,” 2010. [Online]. Available: http://www.opcfoundation.org/

Appendix A: Create MonitoredItem Service Implementation

Source code of Create MonitoredItem Service implementation within SubscriptionServiceHandler is presented below.

```
@Override
protected void createMonitoredItems(ServiceContext serviceContext,
    CreateMonitoredItemsRequest request,
    CreateMonitoredItemsResponse response,
    List<ReadValueId> itemsToRead, List<MonitoredDataItem> items)
    throws ServiceException {

  NodeId nodeId = request.getItemsToCreate()[0].getItemToMonitor()
      .getNodeId();

  // use gateway logic if gateway client contains the nodeId
  if (myGateway.hasNodeId(nodeId)) {

    MonitoredItemCreateRequest[] itemsToCreate = request
        .getItemsToCreate();
    checkRequestLength(itemsToCreate);
    // throws Bad_SubscriptionIdInvalid

    UnsignedInteger clientIndex = null;
    Subscription serverSubscription = getSessionManager().getServer()
        .getSubscriptionManager()
        .getSubscription(request.getSubscriptionId());

    // create new subscription, if subscription not created for nodeid
    for (int i = 0; i < itemsToCreate.length; i++) {
      nodeId = itemsToCreate[i].getItemToMonitor().getNodeId();
      clientIndex = myGateway.getClientIndex(nodeId);
      if (myGateway.getClientSubscriptionId(clientIndex,
          serverSubscription.getSubscriptionId()) == null) {
        try {
          com.prosysopc.ua.client.Subscription clientSub = myGateway
              .getClient(clientIndex).addSubscription(
                  myGateway.createSubscription(Integer
                      .parseInt(clientIndex
                          .toString())));
          myGateway.putIntoSubscriptionMap(
              serverSubscription.getSubscriptionId(),
              clientIndex, clientSub.getSubscriptionId());
    } catch (StatusException e) {
      e.printStackTrace();
    }
  }
}

UnsignedInteger clientSubscriptionId = myGateway
    .getClientSubscriptionId(clientIndex,
        serverSubscription.getSubscriptionId());

final TimestampsToReturn timestampsToReturn = request
    .getTimestampsToReturn();

// initialize result and diagnostic info structures
MonitoredItemCreateResult[] results =
  new MonitoredItemCreateResult[itemsToCreate.length];
DiagnosticInfo[] diagnostics = new DiagnosticInfo[itemsToCreate.length];
// go through all the items needed to be created
for (int i = 0; i < itemsToCreate.length; i++) {
  try {
    results[i] = new MonitoredItemCreateResult();

    NodeId itemNodeId = itemsToCreate[i].getItemToMonitor()
        .getNodeId();

    // get necessary gateway client and namespace indices
    // along with subscriptionId
    Integer[] clientIndices = myGateway
        .getClientIndices(itemNodeId);
    clientIndex = UnsignedInteger
        .parseUnsignedInteger(clientIndices[0].toString());
    clientSubscriptionId = myGateway.getClientSubscriptionId(
        clientIndex, request.getSubscriptionId());

    // create item in the gateway server
    MonitoredItem item = getSessionManager()
        .getServer()
        .getSubscriptionManager()
        .createMonitoredItem(serviceContext,
            serverSubscription, timestampsToReturn,
            itemsToCreate[i]);

    // set nodeId to comply with namespace in underlying server
    itemsToCreate[i].getItemToMonitor().setNodeId(
        NodeId.get(itemNodeId.getIdType(),
        clientIndices[1], itemNodeId.getValue()));

// create item in the gateway client
UnsignedInteger clientMonitoredItemId = myGateway
    .createMonitoredItem(
        myGateway.getClient(clientIndex)
            .getSubscriptionById(
                clientSubscriptionId),
        itemsToCreate[i].getItemToMonitor()
            .getNodeId(), itemsToCreate[i]
            .getItemToMonitor()
            .getAttributeId(),
        item.getFilter(), itemNodeId);

if (clientMonitoredItemId == null) {
  results[i].setStatusCode(new StatusCode(
      StatusCodes.Bad_NodeIdExists));
} else {
  // ensure that new items are updated to item hashmaps
  // so they can be linked to gateway client itemid
  myGateway.putIntoMonitoredItemIdMap(clientIndex,
      clientSubscriptionId, clientMonitoredItemId,
      item.getMonitoredItemId());

  // set results to show to the outside client
  results[i].setRevisedQueueSize(UnsignedInteger
      .valueOf(item.getQueueSize()));
  results[i].setRevisedSamplingInterval(item
      .getSamplingInterval());
  results[i].setStatusCode(StatusCode.GOOD);
  results[i]
      .setMonitoredItemId(item.getMonitoredItemId());

  final MonitoringFilterResult filterResult = item
      .getFilterResult();
  // set filter to item
  if (filterResult != null) {
    results[i].setFilterResult(ExtensionObject
        .binaryEncode(filterResult));
  }
  // finally add item to monitoredItems list,
  if (item instanceof MonitoredDataItem) {
    itemsToRead
        .add(itemsToCreate[i].getItemToMonitor());
    items.add((MonitoredDataItem) item);
```

```
     }
    }
   } catch (ServiceException e) {
    getLogger().debug(e);
    results[i].setStatusCode(e.getServiceResult());
    diagnostics[i] = e.getDiagnosticInfo();
   } catch (StatusException e) {
    getLogger().debug(e);
    results[i].setStatusCode(e.getStatusCode());
    diagnostics[i] = e.getDiagnosticInfo();
   } catch (EncodingException e) {
    results[i].setStatusCode(e.getStatusCode());
    diagnostics[i] = new DiagnosticInfo(e.getMessage(), null,
      null, null, null, null, null);
   }
  }
  // set modified results to response result field
  response.setResults(results);
  // set diagnostic information to response diagnostic field
  response.setDiagnosticInfos(diagnostics);
 } else {
  // use the unmodified functionality, i.e. normal node in gateway server
  super.createMonitoredItems(serviceContext, request, response,
    itemsToRead, items);
 }
}
```

Appendix B: Loading and Storing SQL Commands

Loading of DataItems from SQL schema with JDBC Statements, ResultSet and SQL syntax is presented below.

```
public synchronized void loadDataItems(DateTime startTime, DateTime endTime,
List<DataValue> history, NodeId nodeId) throws SQLException {
  try {
    // get node identifier used in the database
    int node = getDatabaseNode(nodeId);
    // timestamps stored as integer in the database
    long st = startTime.getValue();
    long et = endTime.getValue();
    String cmd = "";
    ResultSet rs;
    // select all values within certain time range into result set
    switch (databaseType) {
    case SQLServer:
      cmd = String.format(
        "SELECT * FROM [dbo].[%s] WHERE ((Node = '%d') AND "
        + "(SourceTimestamp BETWEEN %d AND %d))",
        dataTableName, node, st, et);
      break;
    case mySQL:
      cmd = String.format(
        "SELECT * FROM %s WHERE ((Node = '%d') AND "
        + "(SourceTimestamp BETWEEN %d AND %d))",
        dataTableName, node, st, et);
      break;
    }
    // execute statement into result set
    rs = stat.executeQuery(cmd);
    // process result set
    while (rs.next()) {
      String value = rs.getString("Value");
      String statusCode = rs.getString("StatusCode");
      long sourceTimeStamp = (long) rs.getObject("SourceTimestamp");
      long serverTimeStamp = (long) rs.getObject("ServerTimestamp");
      // store database row into single datavalue
      DataValue dataValue = new DataValue();
      // use stored timestamps
      dataValue.setSourceTimestamp(new DateTime(sourceTimeStamp));
      dataValue.setServerTimestamp(new DateTime(serverTimeStamp));
      // cast value into correct datatype as datatype for NodeId is known
      // set value into value of datavalue
      dataValue.setValue(new Variant(castAsUaType(value, nodeId)));
      // StatusCodes are inherently identified by UnsignedInteger and
      // same format is used in the database
      if (statusCode != null) {
        dataValue.setStatusCode(UnsignedInteger
          .parseUnsignedInteger(statusCode));
      } else {
        dataValue.setStatusCode(value == null ?
          StatusCode.BAD : StatusCode.GOOD);
      }
      // finally add value into history value list used by the HistoryRead
      // service
      history.add(dataValue);
    }
  } catch (SQLException e) {
    e.printStackTrace();
  }
}
```

Saving of DataItems into SQL schema.

```
public synchronized void saveDataItem(DataValue dv, NodeId nodeId)
throws SQLException {
  try {
    // get the serverId used in the database using hashmap implementation
    int node = getDatabaseNode(nodeId);
    // use integer timestamps
    DateTime sourceTimeStamp = dv.getSourceTimestamp();
    DateTime serverTimeStamp = dv.getServerTimestamp();
    long dtSrc = sourceTimeStamp.getValue();
    long dtSrv = serverTimeStamp.getValue();
    // initialize SQL command
    String cmd = "";
    // check if nodeId is new, assumes nodeid_datatype_map
    // is up-to-date, update accordingly
    if (!(nodeId_to_datatype.containsKey(nodeId))) {
      saveNodeId(nodeId);
      updateNodeIdDatatypeMap();
    }

    // use different commands regarding database in use
    switch (databaseType) {
    case SQLServer:
      cmd = String.format(
          "INSERT INTO [dbo].[%s] VALUES ('%d','%d','%d','%s','%s')",
          dataTableName, node, dtSrc, dtSrv,
          dv.getStatusCode().getValue().toString(),dv.getValue());
        break;
      case mySQL:
        cmd = String.format(
          "INSERT INTO %s VALUES ('%d','%d','%d','%s','%s')",
          dataTableName, node, dtSrc, dtSrv,
          dv.getStatusCode().getValue().toString(),dv.getValue());
        break;
      }
      // update value table
      stat.executeUpdate(cmd);
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
```

Appendix C: History Gateway Data Type Casting

Type casting procedure in History Gateway database implementation.

```
private Object castAsUaType(String value, NodeId nodeId) {
  String datatype = nodeId_to_datatype.get(nodeId);

  if (datatype.equals("Boolean")) {
    return Boolean.valueOf(value);
  }
  if (datatype.equals("Byte")) {
    return Byte.valueOf(value);
  }
  if (datatype.equals("UnsignedByte")) {
    return UnsignedByte.valueOf(Integer.parseInt(value));
  }
  if (datatype.equals("Short")) {
    return Short.valueOf(value);
  }
  if (datatype.equals("UnsignedShort")) {
    return UnsignedShort.valueOf(Integer.parseInt(value));
  }
  if (datatype.equals("Integer")) {
    return Integer.valueOf(value);
  }
  if (datatype.equals("UnsignedInteger")) {
    return UnsignedInteger.valueOf(Long.parseLong(value));
  }
  if (datatype.equals("Long")) {
    return Long.valueOf(Long.parseLong(value));
  }
  if (datatype.equals("UnsignedLong")) {
    return UnsignedLong.valueOf(Long.parseLong(value));
  }
  if (datatype.equals("Float")) {
    return Float.valueOf(value);
  }
  if (datatype.equals("Double")) {
    return Double.valueOf(value);
  }
  if (datatype.equals("String")) {
    return value;
  }
  return null;
}
```