TU Clausthal

Clausthal University of Technology

A Data-Centric Information Architecture for Power Systems

Doctoral Thesis
(Dissertation)

to be awarded the degree of
Doctor rerum naturalium (Dr. rer. nat.)

Submitted by
Christoph Gerdes

approved by the Faculty of Mathematics/Computer Science and Mechanical Engineering, Clausthal University of Technology

Supervised by:

Prof. Dr. Jörg P. Müller
Prof. Dr. Sven Hartmann

Date of oral examination
TBA

EIDESSTATTLICHE ERKLÄRUNGEN

Hiermit erkläre ich an Eides Statt, dass ich die bei der Fakultät für Mathematik/Informatik und Maschinenbau der Technischen Universität Clausthal eingereichte Dissertation selbständig und ohne unerlaubte Hilfe angefertigt habe.

Die benutzten Hilfsmittel sind vollständig angegeben.

Christoph Gerdes

Hiermit erkläre ich an Eides Statt, dass ich bisher noch keinen Promotionsversuch unternommen habe.

Christoph Gerdes

Abstract

Over the last decades, changes in *society*, *economy* and *technology* have let to the emergence of new requirements for industrial systems [16] [12]. In the near future, new kinds of systems will appear that differ from their precursers by their inherent organisation and the kind and number of system entities.

In particular the size and heterogeneity of the new systems give rise to a high level complexity. In contrast to large-scale systems in other domains, e.g. media or communication, the industrial domain has particular (non-functional) requirements, e.g. stability and safety, that must be addressed essentially.

In this work, a data-centric information architecture for large-scale industrial systems is developed to address the complexity issues. The architectural approach allows for a system model which is not top down engineered but rather implements a bottom-up design. Based on scenarios from present power systems, key requirements are identified and used to validate structures, methods and algorithms of the architectural design.

This thesis makes the following scientific contributions:

- *Concept*. The concept of the Service Ecosystem for Energy Services provides a model for the creation and operation of large-scale power infrastructures. It provides methods for collaboration, communication and exchange. This concept for power infrastructures can be utilised also for other types of large-scale industrial systems.
- *Architecture*. Motivation and elaboration of a data-centric architecture as foundation for distributed intelligence for large power networks. Closing the gap between lower level networking and control applications this approach lays the foundation for advanced control algorithms required, e.g., in de-central power generation context.
- *Data model*. A minimalist canonical data model is developed which allows for communication and co-operation of individual entities.
- *Programming language*. The Service eCoSystem Query Language (SCSQL) enables the declarative specification of information flows between entities. Furthermore large volume data streams can be pre-processed in-network reducing utilisation of the communication infrastructure.

- *Peer-to-Peer protocols*. Transition of P2P protocols originally developed for data sharing in the Internet to the energy automation domain for monitoring and control purposes.

Acknowledgements

This work would not have been possible without the help and support of many people. I would like to thank my supervisors at university, Prof. Dr. Jörg P. Müller and Prof. Dr. Sven Hartmann for their support and valuable feedback. Furthermore, I would like to thank the current and former members of the peer-to-peer and grid computing program at Siemens Corporate Technology for their support and enlightening discussions, namely Dr. Kolja Eger, Christian Kleegrewe, Stephan Merk, Sebnem Rusitschka, Alan Southall and Dr. Gerd Völksen. My thanks also go to my former department head Dr. Burghardt Schallenberger and division head Prof. Dr. Hartmut Raffler for supporting my publications and conference participations. When presenting my results at scientific conferences, I noticed excellent contributions by other researchers that chose a similar domain for the motiviation of their work. However, often the motivations of these contributions suspected that their problem to solve is rather hypothetical and does not appear as such in the real world. My personal requirement for this thesis was to create solutions for real-world problems. Numberous discussions with colleagues from the Siemens Sector Energy helped me to identify the key research challenges. Moreover the close cooperation with the business units allowed me to validate the concepts of the thesis on concrete scenarios and hardware. Particulary, I would like to thank Gerhard Lang, Dr. Götz Neumann and sector CTO Dr. Michael Weinhold.

The work on this thesis continued over a five year period. It took many nights, weekends and holidays to develop concepts, implement simulations, present the results to the scientifc community, and write the final report. Without the support of my family I would not have been able to complete this thesis. I thank my parents and particularly my wife Lisa Ann for support, tolerance, motivation, and out of the box perspectives.

Contents

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts: they'll be obvious

Fred Brooks, The Mythical Man-Month

Chapter 1

Introduction

Over the last decades changes in *society*, *economy* and *technology* have let to the emergence of new requirements for industrial systems [16] [12]. In the near future, new kinds of systems will appear that differ from their precursors by their inherent organisation and the kind and number of system entities.

Daniel Bell describes in "The Coming of Post-Industrial Society", the transition from a manufacturing based economy towards an economics of information [13]. Instead of cultivating farms, producing steal or mining coal, post-industrial societies generate educated people and large organisations. Doubtlessly his vision became reality in the late eighties and nineties of the last century when business conducted in the finance sector grew beyond worldwide exports [153] [47]. The transition of society was fostered by the emergence of digital systems like the personal computer and the Internet which enabled knowledge sharing and collaboration at a global scale.

Information technology started to digitise manufacturing systems to achieve higher efficiencies and support new production management strategies such as *built to order* [57] and *mass customisation* [109]. Driven by the new needs of the information society and the capabilities of modern information technology, industrial systems have been evolving from central assembly-lines towards dispersed, digital and automated batch production systems.

However, in order to construct and operate these new industrial systems, several conflicting requirements need to be balanced. Globalisation and growing competition among market players require extended differentiation. Customers demand products that precisely match their individual needs. Individual production, however, conflicts with increased pricing pressure as a result of competition among players. The production of customised goods boosts the complexity and dynamics of product management, production and mar-

keting. Hence new systems must offer both the efficiency of automated mass-production and the delivery of customer specific products. Moreover, the change in production processes demands co-operation among autonomously acting players, the ability to quickly react to evolving requirements as well as the integration of third parties, e.g., the consumer into the production process. Consequently, a recent McKinsey report [31] states that, in order to gain market share, corporations require information and communication architectures with the flexibility to quickly react to changes in demand, the support of agile implementation of new business models and the ability to seamlessly co-operate with third parties.

The Internet fosters rapid development of Information and Communication Technologies (ICT) to create computer networks that connect millions of people world wide. Originally developed for scientific and consumer applications, in the last decade, standardised communication technologies influenced industrial applications as well. However, a variety of challenges explicitly the *complexity* [16] [12] [29] of the new technology hinders thorough adoption of ICT in industrial systems.

In recent years two parallel worlds established. The world of business processes and office applications versus the automation world with control equipment, data acquisition systems and engineering tools. Both worlds are largely isolated silos without unified and integrated interfaces [70]. The separation is not only highly inefficient, but also inhibits thorough implementation of cross-enterprise business processes and hence limits partner co-operation. However, even with vertically integrated automation and business systems, new challenges arise due to the sheer size of systems. These large-scale systems consist of thousands of heterogeneous platforms, controllers, sensors and actors connected through heterogeneous wire-line and wireless networks. The challenges inherent to these kind of systems are not limited to technical issues like huge code repositories with millions of lines of code or very large data volumes, but also include non-technical issues like the large number and diversity of people involved in using as well as creating these systems. Individual parts of the system will be owned and maintained by different players with, potentially, conflicting business goals. To overcome these challenges, new architectures for large-scale industrial systems are required.

In this work, a data-centric Information architecture for large-scale industrial systems is developed by the example of a large-scale power system. The architectural approach allows for a system model which is not top down engineered but rather implements a bottom-up design. Based on scenarios from present power systems, key requirements are identified and used to validate structures, methods and algorithms of the architectural design.

This introductory chapter is structured as follows: First, the systems under investigation are introduced. The research questions and challenges addressed in this work are extracted by analysing architectural approaches and requirements of the target systems. Subsequently, the research hypothesis is formulated. The chapter concludes with a summary of contributions of this thesis and outline of this thesis.

1.1 Large-scale Industrial Systems

This work targets large-scale industrial systems. In compliance with general systems theory, a system is understood as being comprised of a set of interacting or interdependent entities forming an integrated whole [154]. The particular systems investigated in context of this work, are characterised by their size and inherent complexity on the one hand, and by their critical requirements in regard to safety and reliability on the other. The systems are open, meaning that they can be accessed, manipulated and extended by computers, people and organisations. Instead of a formal system definition, the following summarises the key characteristics of the systems under investigation. The definitions are used throughout this work.

1.1.1 Large-Scale System

A large-scale system consists of thousands of entities connected by some kind of network. Entities are controlled by at least two different parties. The system executes several processes in which one or more entities may be involved. Successful operation of the system may depend upon collaboration of entities as well as their controlling stakeholders. The emerging properties of a large-scale system are usually not obvious from the properties of individual entities. Due to the size of the system, a globally exact system state cannot be assessed. An example of a large-scale system is the *Internet* which is comprised of millions of nodes each operated by a different legal entity. Another example are product lifecycle management (PLM) systems which co-ordinate product development and production processes on a cross-organisational scale. PLM systems often manage entire supply chains with a multitude of different legal entities.

1.1.2 Industrial System

An industrial system consists of multiple entities (sensors, controllers, actors) engaged in a controlled environment to achieve a specific production goal. Entities are manufactured to

reliably work even under extreme conditions, e.g. heat, dust, radiation. Hard- or software failures of individual entities may cause physical damage to equipment and threat life or health of personnel.

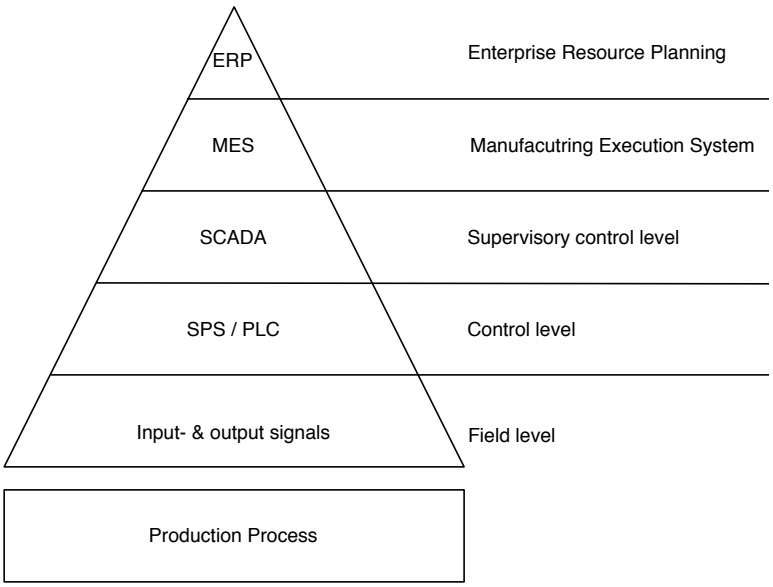


Figure 1.1: The automation pyramid. A layered model of an industrial automation system.

Industrial systems are often structured according to a multi layered model frequently referred to as *automation pyramid* (Figure 1.1). At the top levels, business applications like Enterprise Resource Planning (ERP) and Manufacturing Execution Systems (MES) visualise and process data from the factory floor. Supervisory Control And Data Acquisition (SCADA) systems collect data and provide Human Machine Interfaces (HMI). At the lower levels, actuators and sensors are controlled by, e.g., Programmable Logic Controllers (PLC) which directly control the production processes. The pyramid shape is due to the fact that number of devices and vendors in lower levels exceed those in higher levels. Recently, e.g. by Vogel-Heuser et. al. [146], this model has been found to be no longer accurate. For instance, as the capabilities of sensor hardware increases, intelligence migrates to lower levels. Additionally, technologies like industrial Ethernet allow for direct access from MES to the field level thereby bypassing the control level.

In context of this work, industrial systems are not restricted to the automation pyramid model. Rather the structure of the system is assumed to emerge dynamically based on the qualitative description of system entities as will be explained in Chapter 6. This approach allows for a system model independent of the physical details of the underlying communication infrastructure.

1.1.3 Example: Large-scale Power Systems

Power infrastructures are an example of large-scale industrial systems. As a matter of fact, power systems belong to the class of the largest machines built by man. They are absolutely critical for modern societies. A power system consists of thousands of devices ranging from power generation via transmission and distribution to consumption. Power systems are highly connected with other power systems operated by separate legal entities. Hence considerable communication and co-ordination effort is required to guarantee stable operation. Today power systems are in profound change. Driven by regulatory and economic factors [152], power system operators must adopt new business models like individual tariffs and value-addded services in order to remain competitive [147]. The following section introduces the state of power systems today and provides an outlook to the fundamental change that will occur in the near future.

1.2 Motivation: Towards a new Communication Architecture for Power Infrastructures

Electrical energy constitutes the backbone of modern society and therefore is of utter importance for all residential, commercial, industrial and transportation infrastructures. Global pressure to reduce green house gases as well as ongoing unbundling and liberalisation of energy markets together with the emergence of new and disruptive business models, make power systems an urgent area of research. Similarly to other industries, e.g., telecommunication [148], liberalisation [152] starts to transition energy generation and distribution into competitive and cross-organisational business.

1.2.1 Electric Power Infrastructures Today

Power infrastructures belong to the largest, most complex and therefore most difficult to control machines humans have ever built. The peculiarity of power systems is based on a set of technical and physical characteristics. Among the most significant are: first, the fundamental properties of electricity itself which make power systems difficult to control. Electricity travels at nearly the speed of light. This means that any consumer connected to the grid has instantaneous effect on the network since for every electron “consumed”, a new one needs to be injected into the grid in real-time. Second, today electricity cannot be stored or can only be stored at a very limited scale. As a result, network operators must maintain almost exact balance of generation and consumption at all times. Third,

the power grid is highly complex. Power received by a consumer cannot be traced back to a single generator. In the past, in many regions of the world the grid grew continuously thereby integrating small local grids into the main grid. Therefore, today a labyrinth of paths exists between generator and consumer. If one transmission line fails, its load is automatically rerouted through alternative lines. If these lines operate already at their maximum capacity, they will overload and shutdown as well, causing cascading effects and region wide blackouts. The power grid therefore requires constant supervision and control in order to maintain stable operation and quality of service.

Traditionally, power generation was fully decentralised. Spread over large geographic areas, isolated generators and small networks provided electrical energy to cities and industrial plants. Power quality ([126] p. 45ff.), i.e., constant frequency and voltage, was considerably below today's standards with frequent blackouts and fluctuations in voltage and frequency. To increase quality and efficiency, in the first half of the last century, isolated networks and generators were gradually centralised and nationalised. Today, electricity networks are strictly hierarchical with a top-down flow of electric power from a few large power plants down to a large number of consumers. Power is generated usually at 11-25KV at the power plant and then stepped up to 200-400KV for transmission over long distances. Transformers at substations transform power to 110KV for industrial consumers or small scale power plants. At load centers such as citites, power is again transformed to 1-50KV before it passes the final transformation to 220V to reach the end customer. The main challenge, keeping generation and consumption in balance, is a complex control task that relies on accurate measurements or estimates of the current network state at all voltage levels. Today, however, power networks are partially black boxes due to the lack of sensor equipment and communication infrastructure. Hence, load efficiency is sub-optimal and the flow of power is often unknown. However, based on static models, flows and states can be estimated and, at least, robust operation can be achieved.

1.2.2 Future Electric Power Infrastructures

The situation is changing dramatically as power generation shifts again from a central structure towards distributed generation. With the increased usage of renewable energy sources such as wind, sun and geothermal sources, former consumers now feed electrical energy into the distribution network. Consequently, power flows reverse, flowing from bottom to top, causing unforeseen dynamics and unstable operation. Due to the stochastic[1] behaviour of the sources and the modern highly meshed electricity network, the situation

[1]The output of renewable sources often cannot be precisely forecasted due to non-deterministic environmental influences, e.g wind or sun intensity.

is fundamentally different from the de-central generation scenario of the first part of the last century.

While power systems are capable to compensate a certain degree of de-central feeders, once de-central generation reaches a critical threshold, new communication and control infrastructures are needed to maintain continuous supply of power. The static estimation models will no longer deliver accurate results such that the availability of real-time information on the grid status becomes a mandatory requirement for stable control decisions. To retrieve an accurate snapshot of the system, information must be sampled at high rates at thousands of nodes, yielding high volume streams of data that need to be collected and processed in a timely manner.

Data sources are highly heterogeneous with many hardware platforms and different communication standards operating concurrently. Sources are not restricted to sensors and automation equipment, but rather the diversity of data reaches from meteorological data, power measurements, temperatures of power lines and equipment to pricing information and other economic data.

The challenges cannot be addressed individually by electric utilities but rather a coordinated effort is required. Electrical networks of different utilities are highly connected and thus cannot be looked at in isolation. Actions taken by one utility, e.g. shutdown of a high voltage line, has immediate effects on the networks of other utilities. It is therefore mandatory that utilities provide efficient and transparent communication endpoints for cross-enterprise data exchange.

1.3 Technical Requirements

In this section, key requirements to further characterise large-scale industrial systems are explained. The analysis is based on the example of power infrastructures as briefly introduced in the previous sections. An in-depth analysis of requirements and tactics to achieve associated quality attributes is conducted in Chapter 6.

- *Performance.* The time required to generate a response to a given stimulus is critical for successful operation of the system. If, based on the sensed situation, a decision is delayed equipment damage or even harm of human life may be the consequence.
- *Predictability.* Entities participating in control applications must behave deterministically. An entity must guarantee to respond within a specified time window. Not meeting this constraint renders the entities' contribution useless as the response cannot be processed and will be discarded.

- *Modifiability.* Automation devices are deployed to operate for decades. The environment and the requirements of a production system, accordingly, may change over time requiring adaption to the new situation. Modification includes deployment of new control algorithms as well as rules to discover and connect to new entities that participate in the system.

- *Security.* Enabling modification of system functionality introduces the possibility of malicious manipulation of entity functionality. Furthermore, opening networks for third parties, e.g. for suppliers or service providers makes entities vulnerable for denial of service attacks which may interrupt the system functionality.

- *Adaptability.* Although industrial devices can be assumed to be very reliable, increasing the number of entities in the system also increases the probability of failure. Individual devices need to adapt to internal as well as external conditions and their dynamic change over time. The quality attribute of *Adaptability* can be further classified as

  - *Scalability.* The ability to adapt to a changing number of entities in the system.
  - *Flexibility.* The adaption of the application to environmental conditions during runtime.
  - *Stability.* The capability of a system to maintain its functionality even in the presence of frequent adaptions.

- *Observability.* Individual entities may undergo complex state evolution when executing processes in the system. Often entities must co-operate to achieve a common goal. Failure detection and state assessment of remote entities rely on open interface to state and performance information.

- *Awareness.* Entities are highly connected. Actions taken by one entity may have effect on other entities. Complex cascading effects may influence system performance and stability. Individual entities must be aware of either the full system state or an estimation thereof, when making control decisions.

- *Availability.* Industrial systems require individual entities to be reliable and highly available. The requirement includes the availability of function as well as data.

- *Integrability.* Entities directly interface with the business process infrastructure of, e.g. a plant operator or utility. Aiming for fully automated processes, seamless integration into the existing IT infrastructure is mandatory.

1.4 Architectural Patterns of Industrial Systems

By examining the infrastructure of industrial systems such as the ones described in previous sections, a variety of architectural approaches can be identified. However, before we can discuss the different architectural patterns, the term *architecture* needs further clarification. Throughout this work[2] the term *information architecture* describes the concept and definition of the structure of an information system [155], e.g. a computer based system. The description is non-technical and focusses on universal schemes for structuring and classifying the interaction of system entities. The term *software architecture* is understood as defined by Bass et. al. [11] which state:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Hence, the term software architecture is more technical while the information architecture describes the system at a level independent of a concrete technical mapping. In order to provide a compact description, the following often uses simply the term *architecture* when it is clear from the context whether an information or software architecture is referred to.

This remainder of this section briefly[3] summarises current approaches applied to industrial automation systems, elaborates their deficiencies in context of the requirements discussed in Section 1.3 and identifies the gap this work is aimed to bridge.

Client-Server

The client-server model is used by most current systems. In a client-server based system, clients send requests to a single server where a respective action is executed and a response is passed back to the client. While being simple and efficient in small, static environments client-server architectures rarely resemble the modelled system structure and, as a centralised approach, lack the flexibility and scalability to support the new requirements described in the previous sections.

Distributed Objects

Distributed objects support models closer to the actual system circumstances. Objects usually represent a physical entity or process of the system. Compared to client-server

[2] Note: the concept of an *architecture* is further classified in the context of this work by Chapter 4

[3] An extensive analysis of requirements is conducted in Chapters 6 and 7

architectures, they are more flexible and scalable. The functionality provided by an object is encapsulated by a well-defined interface. This allows for replacement or extension of the implementation leaving other parts of the system untouched. However, communication in distributed object systems is often synchronous due to the lack of pro-activeness of individual objects.

Service Oriented Architectures

In Service Oriented Architectures (SOA), entities are described by the service, i.e. the function, they provide. Services encapsulate resources which execute the service functions. The service interface provides a consistent view of the resource. Besides the service interface, a service description provides information about the interface as well as qualities of its operation. In order to use a service, a service consumer states his interest in a service query and issues the query at a service registry which maintains all available service descriptions. Once found, the service consumer binds to the service first and then executes the desired functions. A key advantage of the service oriented approach is the different temporal options for service binding. Early binding is referred to at design time when requirements are mapped to service descriptions. Late binding, in contrast, renders a system more flexible as the service consumer binds the service at runtime. Ultra-late binding takes the concept one step further as applications are created by composing services dynamically for each invocation and removing services from the application after they are no longer needed. Complex functionality is implemented by orchestrating several services. To architect loosely coupled systems, service orchestrations are usually specified declaratively in standardised orchestration languages.

Similarly to distributed objects, the request/response communication model of services is synchronous. Services are not expected to forward pro-actively information to other services.

Multi-Agent Systems

The intelligent agents paradigm [96] is a modern approach that has had influence on the industrial sector in the recent past but has not yet reached major applicability. Intelligent agents provide a higher level of autonomy as each agent operates with its own thread of control. Provided with high level communication languages, agents can resemble closely the structure of modelled systems. In the process of modelling industrial automation systems either functional or physical decomposition is utilised [105] [19]. Functional decomposition breaks the system into tasks and subtasks that are each represented by

an agent. In contrast, physical decomposition breaks the system down to physical objects that are represented by agents. Consequently agent models provide intuitive abstractions to industrial systems.

For data intensive applications efficient data exchange is important. For instance, in certain situations a utility might not be interested in individual devices or tasks to perform. Of interest is rather the state and dynamics of the system. The operator needs to know if the system is fully operational, if there exists an error or whether the system operates at maximum efficiency. These kind of queries demand a high level abstraction of the system. For example, the operator cannot know which device might have recorded an error nor is it practical to query all devices. He wants to issue complex queries of the form "show me all regions where voltage has been unstable within the last 20 minutes". In other words, the operator needs a declarative interface to the industrial system, where he can describe what he requires in which quality rather than where from and how.

In an agent architecture this functionality resides between lower level networking and application layer intelligence. As agents are usually hosted by a container or platform, which provide this communication layer, no explicit data centric modelling is undertaken. Therefore, the flexibility of the model is reduced and the methodologies seem to miss an important aspect of the target systems.

Data-Centric Architectures

The architectural patterns and paradigms introduced so far envision software systems as a set of tasks, designed as functions, objects, services or agents. The emphasis on functionality is problematic in data intensive domains [61]. Complex data models and non-functional data qualities often have considerable influence on functional interfaces. Hence, if not considered at an early phase in the design process, severe design flaws may be the result.

Data-centric approaches promote data to a first class citizen in the architecture. In the data-centric approach, a system is modelled by defining data elements that describe the components and characteristics of the system. In other words, focus is set on data flow and transformation within system, rather than the processes, i.e. functions, that perform these actions. As one of the first steps in designing a data-centric architecture, the architect creates a *domain model* by answering the following questions: what are the entities of the systems and what kind of data do they consume, generate or transform? What qualities are associated with the data? In succeeding steps, the domain model is mapped to concrete data structures and types. Eventually, the data structures are mapped to implementation artefacts.

In heterogeneous scenarios several domain models may exist. In order to enable communication and exchange of data between functional entities a canonical data model, i.e. a *lingua franca* is designed. Hence individual components can keep on using their internal domain specific data model. Translators accomplish the conversion between the internal and the canonical data models. The approach gurantees application flexibility and ensures lose coupling between components.

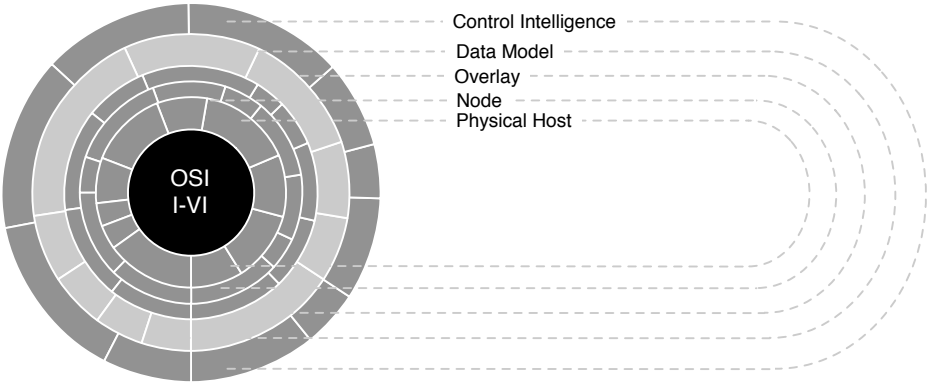


Figure 1.2: Data-centric architecture based on the Open System Interconnection Reference model (OSI)

Figure 1.2 illustrates a data-centric architecture for a large-scale distributed system. At the outer layer, components of higher level applications, such as MES or ERP software are defined. A shared data model describes the characteristics of inter-component communication. The data model is mapped to concrete data structures, e.g. overlays with distributed hashtables. Finally data structures are associated with software components that represented physical hardware devices. Standard networking techniques such as the Internet Protocol (IP) implement the lower level communication infrastructure.

The newly emerging industrial systems constitute a challenging field of research. Due to the characteristics of these systems, research requires interdisciplinary approach that includes both technical as well as social sciences. The research area of so called *Ultra Large-Scale Systems* (ULS) [98] could be a suitable starting point as well for research on large-scale industrial systems. Following the line of argumentation in [98] p. 21ff., the subsequent section elaborates the research questions and challenges that will be addressed by this work.

1.5 Challenges and Research Questions

The complexity of traditional power systems is further amplified by regulatory policies, e.g. liberalisation and unbundling [152]. Future power systems will be characterised by thousands of platforms, sensors, devices and software systems, each controlled and maintained by autonomously acting business players and connected via heterogeneous networks. Correspondingly, new systems will have more actors using them for different purposes. The amount of data will increase dramatically; the number of connections and interdependencies between software components which will be based on an increased number of hardware components will multiply. These systems will not be designed by an individual company but rather developed by multiple stakeholders with, potentially, conflicting needs as well as diverse and evolving requirements. In the following, the challenges for building and operating such systems are identified and grouped into three categories namely: *Design and Evolution*, *Coordination and Control* and *Monitoring and Assessment*.

1.5.1 Design and Evolution

The traditional approach to system design in the context of many individual contributions is by standardisation and development of reference models. This process is time consuming, not agile and agnostic to change and evolution. With individual contributors pursuing non-technical, i.e., strategic agendas in the standardisation consortium the efficiency of the process is often sub-optimal especially with increasing size of the consortium.

The design of large-scale industrial systems calls for new methods beyond standardisation. Extending the technological perspective on system design with economic, legal and social considerations will yield a decentralised design process closely oriented to the goals of the individual contributor while being founded upon a lowest common denominator. Such processes are already in practice as can be observed in several *open-source* projects, e.g. JBoss application server [44] or the Linux kernel (www.kernel.org).

Due to the size of a large-scale industrial system, traditional engineering approaches seem less suitable. In the context of designing these new kinds of power systems, the following questions need to be addressed: How can a system be designed that addresses all individual needs of its users and contributors? How can the system designed be evolved and adapted to changing policies and requirements?

1.5.2 Coordination and Control

Having large numbers of autonomously acting players concurrently using the system, calls for techniques to harmonise individual requirements and ensure achievement of overall system goals. Entities in the system may interact with each other directly or indirectly at design time or during runtime. Coordination and control are required to integrate design and development efforts as well as ensure stable and fair resource utilisation. Interactions may change over time, requiring mechanisms to be able to adapt patterns of coordination and cooperation.

Key questions to be answered in the context of the coordination and control challenges are: How can efficient resource sharing be implemented? How can the system be modified to adapt to new requirements or changes in the environment without considerable interruptions? How can users customise their interaction with other users?

1.5.3 Monitoring and Assessment

During operation the system state may consist of thousands or millions of individual entity states. Using an incomplete or wrong runtime model of the system may yield unwanted responses to control and co-ordination efforts. As an example, consider the management of shared resources, like a CPU or memory. If a resource state is unknown or incorrect, e.g. the measurement of it's utilisation, certain resources may overload while others are not utilised at all. Moreover, if load is shifted between two resources, constant reallocations may introduce unnecessary overhead.

Assessing the state of a highly complex system, however, is not trivial. States emerge globally distributed and may evolve quickly over time. Component interdependencies cause complex transient states. Consequently, complete state assessments are impossible in large scale systems. Advanced methods are required to provide meaningful indicators of the system behaviour.

Key questions to be answered are: What are meaningful indicators that characterise the current system state? How can be determined what effect a control action will have? Since complete state assessment in not possible and hence information is imprecise and uncertain, how do monitoring and assessment mechanisms cope with the constantly and quickly evolving states?

1.6 Hypothesis

In previous sections the type of system that is in the focus of this work was characterised. Key challenges and requirements for large-scale industrial systems of the near future were introduced. Examining existing architectural patterns, several deficiencies were identified that hinder achievement of requirements. Large-scale industrial systems pose considerable challenges for information architectures. The aim of this work is to elaborate an architecture consisting of structures, interaction models and algorithms which addresses the challenges and answers the research questions stated above. Taking all previous sections into consideration, the hypothesis made in this work is summarised as:

> Using future power infrastructures as reference system, the research problems emerging in order to create, operate and maintain large-scale industrial systems are addressed by the design of an open information architecture called: *Ecosystem for Energy Services*. As lowest common denominator for all interacting entities, it builds upon semi-structured data augmented with quality attributes. The architecture enables all actors to interact, provide and consume services. Supporting a continuous, decentralised and agile design process, the ecosystem can be adapted by its users to meet new regulatory and individual business requirements.

By establishing a data-centric architecture for large industrial systems the gap between lower level networking and control applications can be closed. Data availability at the right place and time, localises control problems and therefore reduces control system complexity. A global perspective on all data available in a distributed system enables de-coupling of processes from the communication infrastructure and provides the basis for future extensions on the control layer. By providing mechanisms to adapt to network specifics such as declarative specification of information flows, it ensures scalable real-time availability even for large-scale wide area installations. This enables automation of all controllable equipment under one concise paradigm.

1.7 Contributions of this Thesis

This thesis develops several concepts and a concrete information architecture in order to address the challenges identified in previous sections. The following summarises the building blocks and lists the contributions made to scientific conferences, journals and books.

- *Concept.* The concept of the Service Ecosystem for Energy Services provides a model for the creation and operation of large-scale power infrastructures. It provides methods for collaboration, communication and exchange. This concept for power infrastructurs can be utilised also for other types of large-scale industrial systems.

- *Architecture.* Motivation and elaboration of a data-centric architecture as foundation for distributed intelligence for large power networks. Closing the gap between lower level networking and control applications this approach lays the foundation for advanced control algorithms required, e.g., in de-central power generation context.

- *Data model.* A minimalist canonical data model is developed which allows for communication and co-operation of individual entities.

- *Programming language.* The Service eCoSystem Query Language (SCSQL) enables the declarative specification of information flows between entities. Furthermore large volume data streams can be pre-processed in-network reducing utilisation of the communication infrastructure.

- *Peer-to-Peer protocols.* Transition of P2P protocols originally developed for data sharing in the Internet to the energy automation domain for monitoring and control purposes.

1.7.1 Publications

- Christoph Gerdes and Jörg P. Müller. Data-centric Peer-to-Peer Communication in Power Grids. Proceedings of KiVS Global Sensor Networks (GSN09), 2009. [Online]. Available: http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/228.

- Fabian Stäber, Christoph Gerdes, and Jörg P. Müller. A Peer-to-Peer-based Service Infrastructure for Distributed Power Generation. In Proc. of 17th IFAC World Congress, Seoul, Korea, Intl.l Federation of Automatic Control, 2008. [Online]. Available: http://www.ifac-papersonline.net.

- Christoph Gerdes, Christian Kleegrewe, and Jörg P. Müller. Declarative resource dis- covery in distributed automation systems. In I. Troch and F. Breitenecker, editors, MathMod, volume ARGESIM Report, 2009.

- Christoph Gerdes, Christian Kleegrewe, and Jörg P. Müller. Declarative resource dis- covery in distributed automation systems. Simulation News Europe. To appear.

- Christoph Gerdes, Udo Bartlang and Jörg P. Müller. Vertical Information Integration for Cross Enterprise Business Processes in the Energy Domain. In K. Fischer, A. Berre, J. P. Müller, J. Odell, eds. Agent Technologies for Enterprise Interoperability. Lecture Notes in Business Information Processing (LNBIP), Springer-Verlag, pages 1-28, 2009.

- Kolja Eger, Christoph Gerdes, and Sebnem Öztunali. Towards P2P Technologies for the Control of Electrical Power Systems. In P2P 2008: Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing, pages 180-181, Washington, DC, USA, 2008. IEEE Computer Society.

- Christoph Gerdes, Udo Bartlang, and Jörg P. Müller. Decentralised and reliable service infrastructure to enable corporate cloud computing. In Paul Cunnigham and Miriam Cunnigham, editors, Collaboration and the Knowledge Economy: Issues, Applications and Case Studies, volume 5 of Information and Communication Technologies and the Knowledge Economy, pages 683-690, Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands, October 2008. IIM, IOS Press. Proceedings of eChallenges e-2008 Conference.

- Christoph Gerdes, Christian Kleegrewe, Stephan Merk, Kolja Eger, and Jörg P. Müller. Automation Grid. In proceedings of VDI Kongress AUTOMATION 2009. VDI Kongress AUTOMATION 2009, Baden-Baden, Juni 2009, pages 25-28, 2009. Full paper on CD.

- Christoph Gerdes, Christian Kleegrewe, Stephan Merk, Kolja Eger, and Jörg P. Müller. Automation Grid. In Automatisierungstechnische Praxis, atp. To appear.

- Vivian Prinz, Florian Fuchs, Peter Ruppel, Christoph Gerdes, and Alan Southall. Adaptive and fault-tolerant service composition in peer-to-peer systems. In DAIS, pages 30 - 43, 2008.

1.7.2 Patents

- Autonomous Database for Large Scale Industrial Systems, pending, 2010, [Germany]

- Gossip-based Distributed Demand Side Management, pending, 2009, [Germany]

- Declarative Control System for Industry Automation, pending, 2008, [Germany]

- Hierarchische Aggregations- und Steuerinfrastruktur auf Peer-to-Peer Basis, pending, 2007, [Germany, Europe, US]

- Verfahren zur Synchronisation in parallelen, diskreten und ereignisorientierten Simulatoren von Peer-to-Peer Systemen, 10 2007 024 900, 2007, patent granted, [Germany]
- Verfahren zur Synchronisation in parallelen, diskreten und ereignisorientierten Simulatoren von Peer-to-Peer-Systemen mit integriertem dynamischen Load Balancing, 10 2007 024 900, patent granted, [Germany]
- Verteilte Komposition und verteilte Ausführung zusammengesetzter Dienste in Peer-to-Peer Netzen, patent pending, 2007, [Germany, Europe, US]
- Execution of a Distributed Transaction with Temporary Inconsistencies, patent granted 10 2006 014 909, 2006 [Germany, Europe, US]
- Self-organizing infrastructure for reliable service execution, patent pending, 2006 [Germany, Europe, US]

1.8 Structure of this Thesis

Figure 1.3 depicts the structure of this work. Starting with this introduction (Chapter 1), Chapters 2 and 3 present the *context* for this work by surveying the current state of the art of technologies for large-scale data-centric systems and describing essential concepts of power systems as well providing an outlook on the Smart Grid. The technologies described in Chapters 2 and 3 are clustered and matched against the characteristics of the Smart Grid. The classification motivates the architectural approach and integrates the contributions of this work into the context of background and research questions.

Before the actual concepts and architecture are described, Chapter 4 presents the scientific methods and tools applied to create and evaluate the contributions.

Chapter 5 explains the *concept* of the Ecosystem for Energy Services. The description includes a definition of the ecosystem and its principles. Several abstract models lay the foundation for a technical implementation of the ecosystem metaphor and its key principles. Chapter 6 presents a *software architecture* implementing the concept previously introduced. In the beginning of the chapter two scenarios of existing systems are identified. By analysing concrete use cases, functional as well as non-functional attributes are extracted. The remainder of the chapter introduces the modular architecture of nodes, the node processing model, query execution and optimisation, the index cloud and consistency models. In a brief section, the main concepts of the SCSQL programming language and the corresponding compiler framework are introduced.

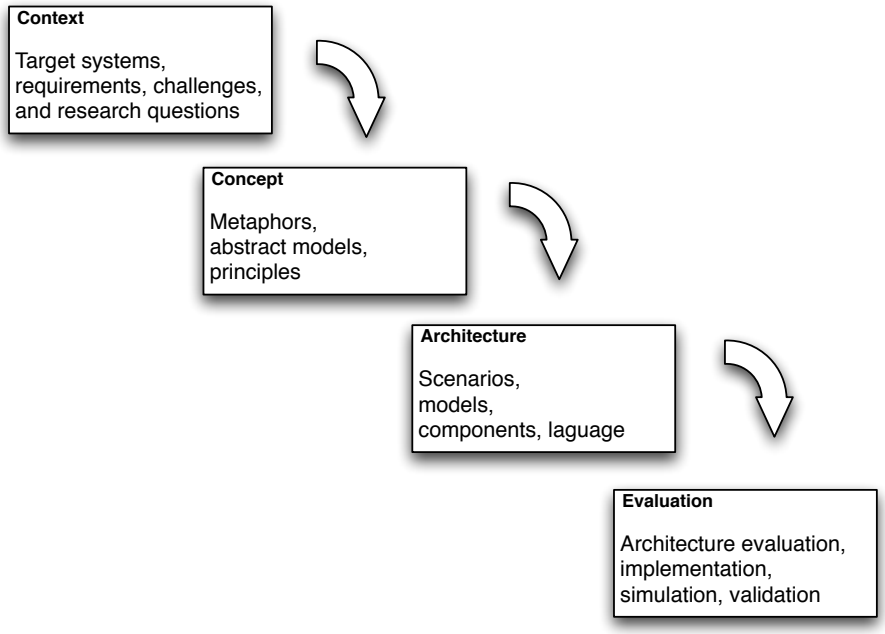


Figure 1.3: Structure of this thesis

Chapter 7 *evaluates* the concepts and architecture described in Chapters 5 and 6 using the evaluation methods described in Chapter 4. The evaluation starts with a qualitative analysis of the architecture and the achievement of quality attributes. In subsequent sections, nodes and index cloud are quantitatively evaluated with the simulation methods described in Chapter 4.

Chapter 8 concludes with a summary and outlook for future work. In the appendix, cost functions as well as the full Enhanced Backus-Naur Form (EBNF) of the SCSQL programming language are provided.

Chapter 2

Technologies for Data-Centric Systems

A considerable body of research has been conducted in the area of data-centric systems. From standard database systems like relational databases (RDBMS), new approaches like highly distributed databases and data stream management systems as well as specialised solutions for large-scale and loosely coupled systems like publish-subscribe systems, Peer-to-Peer (P2P) overlays and application layer multicast systems emerged. More than 30 years of research led to a variety of commercial products and applications.

This chapter presents the current state of the art in data-centric systems. The discussion includes architectural, algorithmic and implementation issues. Additionally, social, environmental and economic effects as well as their architectural influence are reviewed in the context of Ultra Large-scale Systems (ULS).

The contributions of this work extend the current state of the art in data-centric paradigms to large-scale industrial environments. Consequently, the discussion starts with a review of architectural approaches to tackle complex systems. Subsequently, database technology, publish-subscribe, and P2P systems are covered before networking aspects and declarative information collection and dissemination are introduced. The chapter concludes with a summary and alignment with the key contributions of this work.

2.1 Software Architectures for Complex Systems

Already in the year 1968 at the NATO conference [97] in Garmisch-Partenkirchen, Germany, the problem of building software reliably and cost efficient was recognised. The conference led to a new research area and the promotion of a new engineering discipline

which, since then, has been called *software engineering*.

Software architectures constitute a crucial part of system design. Architectures determine the structure of large systems by identifying a set of key components and their relationships. Diagrams and textual descriptions establish a common taxonomy for everyone involved in creating, negotiating and implementing the architecture.

A design process of a large-scale system involves many stakeholders from management, marketing, users, maintenance and customers. Commonly the business relationship to create a software product is that a single party, i.e. the customer, contracts a software company to create a software product according to its needs. Bass et. al. [11] summarise the relationship and the iterative process to create a software architecture in the "Architecture Business Cycle" (Figure 2.1).

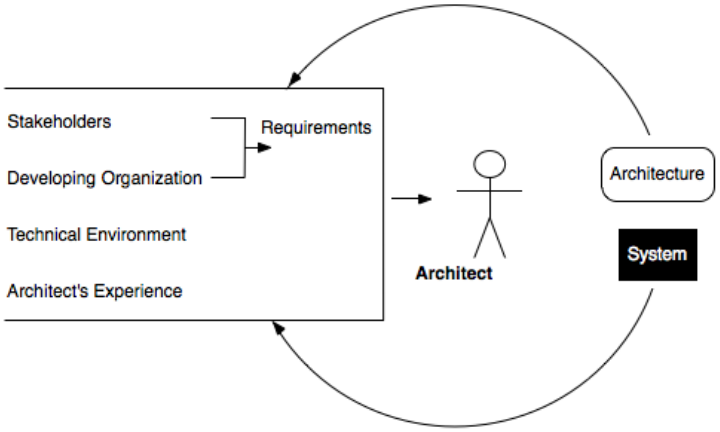


Figure 2.1: The architecture business cycle [11]

The following section provides an introduction to a standardised architecture description thereby defining key terms and concepts. For very large-scale systems, however, the inherent complexity makes the standard design processes less suitable. Ultra large-scale systems (ULS) (Section 2.1.2) paradigms might suggest an alternative to the Architecture Business Cycle, following an open source model.

2.1.1 ISO/IEC 42010 IEEE

The ISO/IEC 42010 standard is a recommended practice for analysis, design, creation and maintenance of software intensive systems. Its conceptual framework shall be the foundation for further discussion on architectural aspects of large-scale systems.

According to IEC 42010, a *system* can be anything from individual applications to subsys-

tems, systems of systems or entire product lines. Systems are located in an environment which may influence the system. The environment determines the boundary and scope of the system.

A system has one or many stakeholders. A stakeholder has *concerns* relative to the system such as performance, reliability or security requirements. The purpose or *mission* is the use the system is designed for, given the environment. Stakeholders may have different roles during the design process. Roles include customer, user, developer, architect and evaluator.

IEC 42010 concentrates on architectural descriptions rather than the actual architecture for a concrete system. An architectural description is organised into architectural *views*. Each *view* targets one or more concerns of the stakeholders. A view captures system properties like process communication, physical deployment or integration. Views are frequently also referred to as *architecture models*.

While the concept of a view is rather abstract and hence can vary between different architectures, a set of standard views are widely accepted. The following lists the most common architectural views providing a brief introduction for each.

- *Conceptual View*: This view point includes all aspects necessary to describe the functional requirements of the software system. The conceptual view point is not bound to any concrete implementation but rather focusses on the entities and their relations within the problem space.

- *Module View*: The module view covers the structure of the software system in terms of modules and respective relations. The organisation of modules is determined by the application of design principles, e.g. information hiding, layering or compiler implied organisation. In contrast to the conceptual view, the module view is tightly coupled with concrete implementations.

- *Process View*: The process view includes non functional requirements e.g. performance or resilience, of the software system. Additionally, it defines how the abstractions of the conceptual view are mapped to processes. A process itself is defined as a set of tasks which can be local or distributed in a network. Processes and tasks constitute an executable module. In other words, the process view describes the interaction of processes and tasks with connectors such as messages, remote procedure calls or events.

- *Physical View*: The physical view describes how software processes are mapped to the execution environment. Particularly non-functional system requirements are

considered. The mapping is determined by the control flow, which was sketched in the conceptual view and is detailed in the process view.

- *View Model*: The four view points above can be described in a so called $4 + 1$ view model [80]. Here, scenarios are used to identify characteristic system entities and their relations. Scenarios are also used for validation of the architecture.

The key roles involved in the architecture description are the architect and the customer. The architect's job is to satisfy the customers requirements by creating or maintaining the architecture. While the role of the architect is not limited to a single person but can be filled by entire teams of architects, the one-to-one business relationship between customer and architect, poses problems in large-scale systems with potentially tens or hundreds of different customer/architect pairs. While the traditional approach is rather centralistic, alternative models like the *open source model* allow de-centralised co-ordination of partners. Consequently, the borders between roles and different phases of the software development process, e.g. analysis, design, implementation, become blurred. The following section illustrates how the research on ULS suggests this model to address the challenges in the design for ultra large-scale systems.

2.1.2 Ultra Large Scale Systems and the Open Source Model

Financed by the U.S. Department of Defense (DoD), Carnegie Mellon University (CMU) researches conducted a study on so called ultra-large-scale systems (ULS). The aim of the study was to establish a research agenda for ULS. An ULS is characterised by consisting of thousands of platforms, decision nodes, weapons, and warfighters which are connected by heterogeneous wired and wireless networks. The challenges inherent with these kind of systems are not limited to technical issues like huge code repositories with millions of lines or code or large volumes of data but also include socio-political issues like the large number and diversity of people involved in using and creating theses systems.

The size of ULS prohibits centralised approaches and calls for de-centralisation starting from planning to development and operation. Large numbers of stakeholders need to co-ordinate their, potentially, conflicting interests, integrate evolving requirements and heterogeneous soft- and hardware. Since with increasing numbers of system entities, failures become the norm rather than an exception, particular attention is required to deploy robust mechanisms to compensate failures.

To tackle the challenges, the report suggests to move from traditional engineering to decentralised design of complex systems. Similarly to cities, where individual houses are

engineered but entire cities are not built by individual organisations, and firms, which are hierarchically engineered but the economy is not, de-central concepts can be applied to ULS. The factors that enable cities and the economy to function are mechanisms to regulate local actions such that global co-ordination can be achieved.

The report uses the metaphor *socio-technical ecosystem* to describe the necessary shift in perspective that is required to build ULS. Similarly to biological ecosystems, where individual organisms compete for resources in complex environments, in socio-technical ecosystems people and organisations compete for limited resources, e.g. budget, bandwidth, storage or sensors. From the report [98]:

> "The concept of an ecosystem connotes complexity, decentralized control, hard-to-predict effects of certain kinds of disruptions, difficulty of monitoring and assessment, and the risks in monocultures, as well as competition with niches, robustness, survivability, adaptability, stability, and health."

One approach to achieve de-centralised, co-ordinated software development has already been proven in numerous projects. With its prominent representatives Linux and JBoss, the open source model (OSM) [116] has been deployed as the development process model in many industry grade projects. In the OSM, source code is freely distributed among stakeholders under an open source license which allows everyone to adopt and modify the source code. Depending on the concrete license, changes to the source code must be shared within the developer community. Hence, the project is driven by self-motivated contributors which, often, are based around the world.

The OSM differentiates itself in several aspects from the traditional software model. Sharma et. al [129] analysed the OSM using a theoretical framework from organisation theory [119]. The work examines the OSM environment along three axes namely *structure*, *culture* and *process* (Figure 2.2). Unlike traditional organisations, OSM communities have de-centralised control and decision making, shared governance and allow free flow of information. The position of an individual within the open source community is solely based on reputation. Individuals are hence encouraged to meet quality requirements to gain reputation.

The open source community is just one example of a de-centralised development model. Others, less code centric, can be found under the umbrella of the "Web 2.0" phenomenon. Here, web-based platforms attract communities that enable individuals to create and share artefacts collaboratively. Examples include social media sites and directories like Facebook. While community participation may be triggered by altruistic and idealistic motivations, modern community platforms enhanced and implement sound business op-

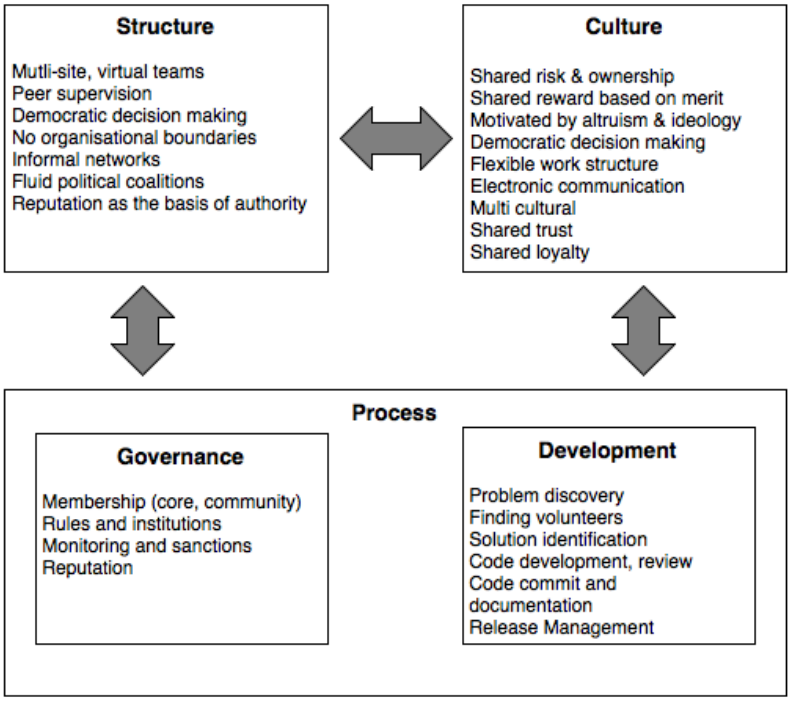


Figure 2.2: The open source software model as examined by Sharma et. al. [129]

portunities. On the one hand, large communities with personal profiles allow targeted advertising and, on the other hand, by opening the platform to independent software vendors and provision of core services like payment and user management, development of additional platform services is motivated which in turn increase the attractiveness of the platform causing yet more individuals to participate. Revenue streams are generated by the service providers as well as the platform providers which earn a share from each transaction between service consumer and service providers.

With this introduction to architectural aspects of large-scale complex systems, in the following, the discussion covers the technical foundation of suitable platforms and open systems. It starts from traditional databases systems and proceeds with advanced data management solutions for large-scale open systems.

2.2 Databases

Databases belong to the first software products available. In 1968, IBM introduced the information management system (IMS), one of the first database systems. Now, nearly four decades later, hundreds of database vendors and systems exist. Databases manage the lifecylce of data, i.e. creation, update, deletion and provide data discovery and search

functionality. Due to their maturity and broad applicability, databases constitute an essential component of a large spectrum of applications.

Databases usually provide strict guarantees regarding Atomicity, Consistency, Integrity and Durability (ACID) of data management. While in the last 20 years Relational Database Management Systems (RDBMS) established as a de facto standard, requirements of recent applications like search engines or web sites with heavy write loads brought up alternative designs with simplified data models. This section starts with a quick review of RDBMS and the relational model. Subsequently, data stream management systems which are often implemented as extensions to traditional RDBMS are introduced. A section on real-time databases introduces current state of the art approaches to an important requirement of the systems investigated in this work.

2.2.1 Relational Database Management Systems

An RDBMS can be found in almost any application from social media to health care, finance, e-commerce and multimedia services. The relational model has been introduced by Codd et. al. in 1970 [27]. It defines a relational algebra consisting of definitions for objects, rules, and operations. A relation, often illustrated as *table*, describes logically connected entities of information. The information is structured by attributes, columns in the table, and a set of tuples, i.e. entities or *rows* of the table. A relation enforces several properties:

- A tuple is unique. There are not two tuples with the same attribute values.
- The sequence of tuples with a relation is not defined.
- The sequence of attributes of a relation is not defined.
- Attribute values are atomic.

Besides these properties, a RDBMS must enforce *entity* and *referential integrity*. Entity integrity ensures that every entity is addressable via a unique *key* whereas referential integrity ensures the integrity of keys between relations. For example, given a relation R1 having a foreign key pointing to the primary key of a relation R2, then the RDMBS ensures that (i) every value of the foreign key in R1 equals the value of a primary key in R2, or (ii) the value of the foreign key is NULL.

The majority of RDBMS available on the market implements the relational model strictly. Occasionally, e.g. in case of MySQL, slight deviations are implemented for the sake of

usability. Architecturally, a typical RDBMS is composed of five major components [62] namely:

1. *Client communication manager*. Depending on the overall application architecture, a database must be able to communicate with a variety of clients, e.g. application servers or transaction monitors. The client communication manager provides the required set of protocols to communicate with different client types. It receives requests from clients and passes them to other components of the database. Moreover, the communication manager transfers generated result sets back to the client and controls access by client authentication.

2. *Process manager*. Once a request has been received, resources to handle the request must be allocated. The process manager manages the internal resources by deciding whether to handle the request immediately or defer it until more resources are available.

3. *Relational query processor*. If the decision is made to execute the request, the relational query processor proceeds by compiling the user query into an *execution plan*. Once compiled, the plan is passed to the plan executor which implements typical relational operators like joins, selections, aggregations, projections and sorting. The query processor manages the interface to lower level storage layers by passing and requesting data records.

4. *Transactional storage manager*. The storage system includes algorithms to manage and access data on physical disks. The storage manager can receive access functions, i.e. *read* as well as data manipulations functions, i.e. *write*, *update*, *delete*. It enforces the ACID properties of the database.

5. *Shared Components and Utility functions*. Once the query has been processed, its results are returned back to the client. Resources are de-allocated and locks are freed. Depending on the size of the result set, it may be transferred gradually back to the client causing multiple invocations of process manager, query processor and storage manager.

Although the relational model has been widely adopted, its strict enforcement of the ACID criteria causes substantial problems for large-scale systems. Consequently, a RDBMS is less suited for the handling of very large volumes of transient data. Applications such as sensor monitoring, finance, trading and network monitoring have in common that they need to process continuous, potentially unbounded sequences of data in a timely manner. The subsequent section introduces data stream management systems which extend standard RDBMSs to handle continuous flows of data.

2.2.2 Data Stream Management Systems

Data stream management systems (DSMS) extend the classic processing model of RDBMSs with capabilities to handle large volumes of transient data also referred to as data *streams*. The characteristics of such a stream based processing model can be summarised as follows:

- Data streams are potentially unbounded sequences of data items, i.e. stream elements, generated at an active data source.
- Stream elements are pushed by the active data source. The DSMS has neither control over the arrival rate nor the order of arrival.
- Stream elements are transmitted only once and are ultimately lost if not explicitly stored.
- Queries over data streams run continuously. New results are produced upon arrival of new stream elements.

DSMS are already utilised in a variety of applications ranging from traffic management to power quality monitoring. A considerable body of research has been conducted in respective research communities. Golab and Özsu provide an overview of current research topics on DSMS in [54]. D. Kucuket et al. [81] introduce a streaming database solution to monitor power quality. Sampled at high frequency, Power Quality (PQ) data [2] grows to large volume already for small installations. The approach is deployed in a scenario covering the Turkish Electricity Transmission System with data sampled at feeders and bus-bars located at transformer substations. Mariposa [139] introduces an architecture for wide area distributed databases. The database features a micro economic paradigm used for query and storage optimisation. AURORA [25], STREAM [48], Cougar [49] and others discuss general query processing in sensor networks. AURORA allows users to create queries in a graphical representation. STREAM and Cougar extend the SQL with temporal semantics.

The wide adoption of DSMS in the industry is proof of their maturity. Most products and projects seem, however, limited to small numbers of nodes and unidirectional flows of data. Despite aggregations, processing is concentrated at central locations. Access methods range from graphical tools to extensions of the SQL. All systems lack functionality to implement user defined distributed functions and the ability to scale to large heterogeneous systems.

Particularly in industrial domains timely processing of data is mandatory. Delayed processing of data may yield equipment damage or harm human life. Real-time capabilities

are rarely found in RDBMS nor DSMS. Therefore, their deployment is often not practical. The following section presents real-time systems, starting with a general definition and standard processing models. Subsequently, real-time databases are introduced as data-centric solutions with real-time capabilities.

2.2.3 Real-Time Data Processing

Timely processing of data is a key requirement for industrial applications. If, for example, an over-current measurement signal is not processed in time, a transformer may be damaged thereby endangering surrounding personnel. The deployment of standard databases on the factory floor is promising for several applications. However, databases are designed for IT environments like offices or data centres. They usually do not provide deterministic resource consumption and generally lack true real-time processing capabilities. However, before the discussion on real-time systems and methods to achieve real-time capable data processing can be started, the concept and notion of "real-time" itself must be introduced.

The term "real-time" is often confused with an operation executing very fast[1], leaving a certain time period for recovery in case of failure or unforeseen delay. Surely, this is not enough in industrial domains because once a function failed to meet a deadline, there might not be a second chance. A widely accepted definition of a real-time system is given in [75]:

> "a computer system in which the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical instant at which these results are produced."

Generally real-time systems are classified as either being *hard* or *soft real-time*. Hard real-time means that a violation of the time constraint yields potentially catastrophic consequences. For instance, timely closing of a valve in a high pressure system requires hard real-time as delayed closure might cause the system to de-compensate. Soft real-time systems can tolerate delays momentarily causing decreased quality of service. For example, a voice over IP (VoIP) system is a soft real-time system where delayed transmission causes decreased audio quality.

Real-time processing capability is achieved by assuring that tasks have instant access to all required resources. Real-time systems typically need to execute more than one task at a time. Tasks may compete for resources such as CPU, memory and network

[1] For example in the context of process control *real-time* is understood as executing faster than the process loop.

bandwidth. Hence a scheduling mechanism is required which invokes a task for a finite amount of time before it passes control to another task. To ensure timely execution, tasks are usually assigned a priority value such that tasks with a higher priority are invoked more often and before tasks with a lower priority. How priorities are assigned to tasks has been an active research area for many years [88]. The main goal of a scheduler is the assignment of priorities such that deadlines of all tasks are met. In [88] this is achieved using scheduling policies i.e. rate-monotonic (RM) and earliest-deadline-first (EDF) for pre-emptive periodic tasks. RM assigns priorities statically according to its period, scheduling tasks with shorter periods first. Aperiodic tasks are scheduled inside a virtual periodic task. EDF sorts at time t all ready tasks according to their deadlines. The task with the deadline closest to t is executed first. RM as well as EDF assume that the worst case execution time (WCET) is known a priori and that tasks are generally pre-emptable. Since finding the optimal schedule becomes a NP-hard problem if more than one resource is shared among tasks, most scheduling algorithms assume the presence of just one shared resource. This is problematic in many concrete use cases as tasks require multiple resources and need to execute critical sections in their entirety and thus are not pre-emptable per se. Consequently, this may lead to an effect commonly referred to as *priority inversion*. Thereby higher priority tasks are blocked because mid priority tasks pre-empt low priority tasks on which the high priority task has a data dependency. Priority inheritance and priority ceiling protocols address the priority inversion problem by analysing the content of the different tasks to identify critical shared sections. For each critical section the task with the highest priority p is found. If a task A with a lower priority enters the critical section, the priority of task A becomes p and thus no task with mid priority can interrupt A. The priority of A is reset once it leaves the critical section. Priority inheritance and ceiling protocols are used in most real-time kernels. However, since they have considerable memory and compute time overhead, they are not suitable in severely resource constraint environments.

The concept of port-based objects (PBO) [137] belongs to the class of time-triggered architectures (TTA) which bring timing constraints directly to the programming model such that compilers can schedule and optimise the software to ensure timing determinism. PBOs interconnect through communication channels thereby forming a global data space with atomic read and write operations. The data space implements a state semantics meaning that values remain valid until overwritten. The activation time of a PBO may vary and the time when inputs are read and outputs are produced may not be regular from activation to activation. The Giotto model [63] extends TTAs in that now both computation as well as communication between tasks are executed time triggered. Each task has predetermined, i.e. at design time, start and end times. Communication is accomplished

between tasks activations. Inputs are obtained at the start time and are available to the system only at the stop time of the execution even if the computation finished earlier.

On the contrary to TTAs, event driven approaches like [89] execute tasks when inputs arrive to fulfil certain constraints predefined by the tasks. Like Giotto, time multitasking (TM) [89] provides predictable input/output timing. TM employs an actor model where actors are equivalent to tasks which also specify execution time and deadlines in terms of trigger conditions. Despite traditional actor models, TM tasks may or may not have their own thread of control. In TM the activation of a task depends either on other tasks or on interrupts. By controlling the time at which outputs are published and by triggering tasks with new events, start and stop times of tasks can be deduced and hence deterministic timing properties are achieved. In TM, tasks represent a sequence of reactions, i.e. finite pieces of computation. Tasks communicate through ports which may have various manifestations, e.g. interrupts, FIFO queues or rendezvous points. Task state and specific data is private within a task. Communication through ports is asynchronous, i.e. interaction does not yield transfer of control flow. Mutual exclusion of reads and writes on ports is still required but tasks can always proceed on their internal state without waiting for other tasks.

In the context of data stream management certain applications, e.g. traffic control systems, surveillance systems or health control systems, require real-time processing and timely availability of information. RTSTREAM [150] provides a query model called PQuery to support soft real-time processing of periodic queries. Once a query is registered with the DSMS, its instances are periodically triggered by the system. Upon initialisation of an instance, a snapshot of the data stream is taken as sole input. New tuples arriving at the system are processed by the next triggered instance. Frequencies and dead lines are specified through extension to CQL and enforced by the execution system. In order to cope with temporary overload RTSTREAM introduces an overload protection mechanism called *data admission*. In the process, data miss ratios (MR) are continuously computed and compared against a predefined target. The difference is passed to a Partial-Integral (PI) controller to generate an admission signal ΔP_{AC} controlling the current admission ratio. The signal ΔP_{AC} is derived by the following equation:

$$\Delta P_{AC} = P_{MR} \times (MR_{ST} - MR_{threshold}) + I_{MR} \times (MR_{LT} - MR_{threshold}) \tag{2.1}$$

where MR_{ST} and MR_{LT} are short and long term miss ratios and $MR_{threshold}$ is the maximum miss ratio defined by the application; P_{MR} and I_{MR} are weights on short and long time miss ratios.

RTSTREAM focusses on periodic queries and assumes that no other types of queries are handled by the query processor. This in somewhat unrealistic as target applications

imply different types of queries i.e. periodic, continuous and snapshot to be executed simultaneously. Additionally query execution time introduces additional unpredictability. In [149] a prediction based QoS management scheme is introduced which features query load estimators by utilising execution time profiles and input data sampling.

It is often argued that hard real-time is needed in the fewest of cases. Most hardware is severely under-utilised leaving large time periods for failure recovery. Considering the fact that non real-time capable hard- and software is offered at considerably lower prices and that most industrial systems work reliably even without hard real-time systems, the argument cannot be discarded entirely. However, real-time support is required at least for safety critical subsystems where failures cannot be tolerated.

Unlike for RDBMS where de-facto standard query languages are established there exists no common standard for real-time systems. Hence, additional complexity is introduced for the application programmer in order to adapt and integrate real-time system into the large-scale system.

2.3 Data Management Beyond The Relational Model

Driven by Internet scale databases of search engines like *Google* and *Yahoo!*, new data models were developed that, although providing similar interfaces, differ substantially from the standard relational model. Google's Bigtable [24] and Yahoo!'s open source equivalent HBase [142] are distributed storages for very large volumes of data. Both are designed to operate on thousands of networked commodity servers. Up to a certain level of detail HBase and Bigtable are equivalent, hence in the following discussion is focussed on Bigtable implying that similar concepts apply for HBase.

The data model is table based with a table being a sparse, distributed, multi-key sorted map, indexed by row key, column key and a time-stamp. Each row key in a table is a string of arbitrary length. Rows are ordered lexicographically by the row key. Partitions of a table, called *tablets*, are dynamically created and may be migrated for distribution and load balancing. Column keys are organised into *column families* wherein data stored in a particular column family is usually of the same type. Column keys are created within a column family and hence rely on their existence. A table may have an unbounded number of columns keys. Column families manage access control as well as disk and memory accounting. Reads and writes for a single row key are atomic, allowing users to implement ACID transaction semantics.

Bigtable is only one part of the Google software stack. Data and log files are stored using the Google File System (GFS) [53] (Hodoop file system (HDFS), is the Yahoo!/Apache counterpart). A cluster management software co-ordinates the shared pool of servers, schedules jobs, monitors resources and manages failed machines. Further, the distributed lock service "Chubby" [17] is used to find tablet servers, to ensure that there is at least one active master server to store access control lists and other concurrent operations. Chubby operates by the provision of a namespace with atomic files and folders to be used as locks. Robustness of the Chubby service is assured by replication, master election and guaranteed replica consistency.

Tablets are distributed in a three-level hierarchy with the location of the root tablet at the first level being a Chubby file. The root tablet contains a special METADATA table which stores the location of all tablets. Tablets are stored in the METADATA table by a row key, consisting of the table identifier and the last row in the tablet.

Although providing a rather general data model, Bigtable and Hbase are tailored for specific problems and environments. Servers are expected to have similar networking capabilities. It is assumed that servers are controlled by trusted entities. Moreover, server infrastructures are assumed to be rather static[2] with few reconfigurations and, due to the controlled environment, Byzantine failure models are out of scope.

While Bigtable, GFS and Chubby constitute a scalable infrastructure for very large data volumes, the MapReduce [32] programming model allows for flexible access and procession of the highly distributed data. The model takes as input a set of key/value pairs and returns a set of key/values pairs as output. Users implement two functions *map* and *reduce* wherein the former creates a set of intermediate key/values pairs which are grouped by an intermediate key I by the framework. The reduce function takes the intermediate key plus a set of values for that key and aims to merge the values to a smaller set referenced by the same key. In the paper the authors provide as example the problem of counting the number of words occurring in a large number of documents to illustrate the paradigm. A possible map function would return the word as key plus an initial count of '1'. The reduce function would sum all for the given intermediate key, i.e. the word and return the list of words together with their count. Although minimalist, the programming model can be applied to a great variety of problems like distributed sorts, full text search, document clustering, machine learning, etc.

Besides the search engine space, new data management architectures emerged from other domains. As Web 2.0 services gained popularity, traditional usage patterns of web services changed. While in a standard Web 1.0 web service, the content was rather static, newer

[2]Compared to not arbitrary nodes in the Internet

services allow the user to create and share their own content. This has significant impact on the read/write performance of data management and storage systems. While for a system that mainly serves static pages, images and videos, the content can be easily replicated and cached, services with a high update or write rate do not scale as well. To maintain consistency, replicas need to be updated synchronously thereby blocking the entire system and degrading performance. Another characteristic of Web 2.0 applications is the sudden increase in traffic, i.e. Slashdot effect, requiring a back-end system to scale very quickly. Often, the strict relational data model and static replication model prevent dynamic scaling causing bad service and the loss of business.

Erlang based Couch DB (http://www.couchdb.org) aims to address these challenges by providing distribution and flexible replication. It does not enforce a schema on the data model and provides a flat address/Id space. Couch DB is a document based database with a MapReduce style interface allowing to query, map, index and filter data using JavaScript. It has a RESTful JSON API that allows it to be accessed entirely via HTTP. The fact not only allows the database to be accessed by a large number of clients and client libraries but also enables the application of standard load balancers and caches.

Couch DB stores documents as JSON objects which consist of field names and values. Values may be strings, numbers, ordered lists and maps. Each object is identified by a unique ID. The database allows to implement so called views, i.e. indexes that provide simple structures for the data. Views are representations of the documents in the database. They are built dynamically by the back-end system. However, since the creation of a view on a database with thousands or millions of documents may be expensive, views are updated incrementally. Due to these delayed updates, the consistency model provided by Couch DB is *eventual consistency*. Views are defined by a JavaScript code which is stored together with a document but do not affect the document itself. Couch DB supports full ACID for document writes and updates. Utilising a multi-version concurrency model, reading clients are never blocked but it is ensured that a client reads the same version of the document from start to end of the read. Documents are indexed in a b-tree using their name and a sequence ID as key. The sequence ID is incremented for each update such that it can be used for conflict resolution on replica merges. Couch DB follows a Peer-to-Peer based, bi-directional replication and synchronisation paradigm. This allows to replicate whole or parts of the database to laptops or servers with low network capacities.

Couch DB is able to handle large numbers of requests, scale quickly and adapt to changing data models. Although replication is Peer-to-Peer based, Couch DB is not designed to run on large clusters with thousands of heterogeneous machines. Its JavaScript based view model allows high flexibility to retrieve and filter data in a distributed manner.

In the following sections the discussion ultimately leaves the space of traditional databases and investigates highly scalable autonomous systems capable to co-ordinate millions of nodes.

2.4 Peer-to-Peer Systems

With the emergence of file sharing services like Napster, Peer-to-Peer (P2P) systems gained popularity as highly scalable and robust distributed systems. While several definitions of P2P systems exist, one of the most comprehensive is found in [136] and reads as:

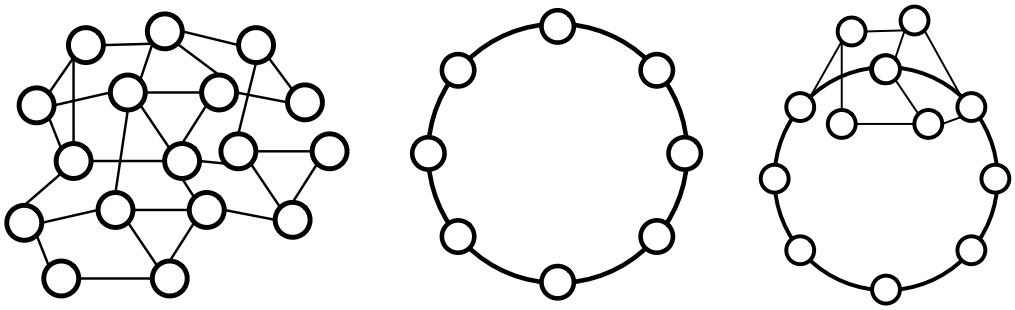
> "A [Peer-to-Peer system is a] self-organizing system of equal, autonomous entities (peers), which aims for the shared usage of distributed resources in a networked environment avoiding central services."

The definition captures the inherent properties of P2P systems, namely: distribution (not limited to de-centralisation), self-organisation, autonomy, i.e. individual peers are under the control of autonomous parties and the facilitation of collaborative resource sharing. P2P systems are implemented as overlay networks and as such provide addressing and routing methods on the application layer of the OSI model. Consequently, they are independent of physical network infrastructures and hence are able to create topologies based on arbitrary criteria, i.e. semantic proximity, geographic location etc..

2.4.1 Structured, Unstructured and Hybrid Networks

P2P systems are often classified as *unstructured*, *structured* and *hybrid* (Figure 2.3). In unstructured networks, nodes organise in arbitrary meshes. Routing tables are randomly built according to any piece of information available. To route data, each node forwards information to all or a subset of known neighbouring nodes. This way, query requests flood the network until a node can satisfy the query or a predefined maximum number of hops is reached. One of the most prominent unstructured networks is the Gnutella network [118]. The maturity and robustness of Gnutella networks led to a large installed base. However, the flooding based routing mechanism hinders ultimate scaling. Unstructured networks like Gnutella generally lack search determinism such that data stored in the network is not guaranteed to be found.

Structured systems, on the other hand, feature deterministic searches and high scalability. Corresponding algorithms enforce a logical topology of the overlay and implement



(a) Unstructured (b) Structured (c) Hybrid

Figure 2.3: Classification of peer-to-peer overlays

structured routing mechanisms. Most structured systems fall into the category of distributed hash tables (DHT) which provide a distribute data structure for de-centralised, self-organising key/value storage. Chord [138], Pastry [124], Tapestry [160] are DHTs which haven been studied in great detail. They enforce ring or tree structures and retrieve data items deterministically with logarithmic complexity (O(log N)). All algorithms utilise binary ordered b*-trees for their searches. A uniform hashing function is applied on the key string to gain a representation in the identifier space. The identifier space is mapped to the peers in correspondence to the topological structure. In a Chord ring, for example, each peer is assigned a portion of the identifier space. Therefor, each peer selects a unique ID at random from the identifier space. Each data item having a hash value greater than or equal to the ID of peer p_i and less then the ID of the following peer p_{i+1} will be stored on p_i.

So called *Content Addressable Networks*, e.g. CAN [114], follow a geometric design to partition a, potentially multidimensional, key space among participating peers. Keys and values are mapped to numerically close nodes. The CAN identifier space can be understood as a n-dimensional Chord key space. For $n = 3$ an identifier has the form $< x, y, z >$ whereas the multidimensional identifiers are gained by partially applying a uniform hashing function, e.g. the first 32 bit map the dimension one, the second 32 bits to dimension two and so on.

In hybrid P2P systems so called *super peers* exist, which accomplish dedicated tasks concerning security and trust, indexing or transactional services. Almost all real-life installations of P2P system, feature slightly modified versions of standard DHTs with support of some sort of central infrastructure, at least for bootstrapping. Version 2 of the Gnutella network has a concept of a super node to handle queries and stabilise the system. Similarly the eDonkey network uses a set of central services for indexing and search. The inherent properties of structured P2P systems, i.e., robustness, determinism, self-organisation make

these systems candidates for solutions for complex industrial systems. Yet, in real-world deployments, the complexity and the corresponding dynamics of structured P2P systems imply considerable challenges. Unstructured systems are generally easier to stabilise and implement, yet provide only a subset of the feature set. Hybrid systems profit from the P2P system characteristics and from the simplicity and stability of central components. Hence they seem ideal candidates for application in the domain of large-scale industrial systems.

While the research on P2P systems has advanced for almost ten years, only few algorithms are used in commercial products today. One reason is the implementation complexity inherent in the algorithms that prevents stable industry grade systems. Another problem are unified access paradigms beyond simple keyword searches, which is discussed in subsequent sections.

2.4.2 Support for Complex Queries

A key challenge for structured as well as unstructured P2P systems is the execution of complex queries, i.e. queries beyond keyword searches and database style queries. Scalability and the lack of query languages seem thereby the major accounts for the search limitations. While most structured systems, such as DHTs, scale very well, i.e. logarithmic with the number of peers, queries are limited to simple keyword or string searches, i.e. "find all items whose names include the given search string". Others, such as Gnutella provide advanced search mechanisms but, due to their inefficiency, non-determinism and flooding, yield often poor results.

In [58] an extension for DHTs to support SQL style query statements is discussed. The approach is not limited to a specific DHT algorithm but starts from a generic interface, i.e. put(key, value), get(key), and extends it with two new functions namely: *lscan*, an iterator to access all objects stored on the local node and a callback *newData* to notify applications that new data has been inserted in the local portion of the DHT. Additionally to the base DHT functionality, peers in this approach are equipped with a query processing layer responsible for providing support for query operators, specifying queries as well as iterating through result sets. Since the flat identifier space of DHTs is not suited to support multiple data structures, e.g. tables, temporary tables, tuples, [58] suggests to partition the flat identifier space into multiple fields each identifying objects of the same granularity. The concepts and ideas developed in [58] let to PIER (Peer-to-Peer Information Exchange and Retrieval) [66] [65] aiming at massively distributed query processing and querying of Internet based data in situ without the need for database design, maintenance or integration. To achieve high scalability, PIER relaxes standard database design

requirements. Consequently PIER sacrifices ACID transactions and provides best-effort results instead.

PIER is build upon a three tier architecture with a DHT as the first tier, the PIER core at the middle layer and applications interacting with the query processor at the top most layer. The underlying DHT is a CAN [115] implementation. The DHT functionality is distributed over three major components, namely: routing, storage and application interface. The routing layer manages the mapping of keywords to specific IP addresses. It maintains local routing tables and reacts to nodes joining or leaving the network. The storage layer is responsible for storing the portion of the data assigned locally to the node. Data is stored either in memory, in standard databases or in the file system. PIER implements a soft state approach meaning that data is stored for a limited time only and must be renewed in order to remain in the system. This mechanism provides a garbage collection feature. This is necessary as peers frequently join and leave the network without deterministic allocation and de-allocation of storage resources. Each data item in PIER has a namespace, a resourceID and an instanceID. The namespace and resourceID are used to compute the hash for the DHT. The instanceID carries some semantic meaning about the data object usually assigned by the user application. When executing a query, PIER contacts peers that hold data in a particular namespace. The application interface provides the scan iterator to support access to all data stored locally. It also provides the hook for the newData callback to notify applications.

The PIER query processor supports selection, projection, distributed joins, grouping and aggregation. However, it lacks a query parser in the current state of implementation; thus queries can be stated through the provided programming interface only. As stated previously, PIER's query processing is not transaction safe. Instead, it provides best effort results. A correct result set is defined as the slightly time-dilated union of local snapshots published by all reachable peers at the time the query was issued. Real-time is provided at none of the modules.

The research surrounding PIER paid particular attention to *join*-operators [66]. Based on symmetric hash joins [156] each peer in namespace N_R or N_S performs an $lscan$ to locate each R and S tuple. Tuples that match to all predicates are then copied and stored in a unique namespace N_Q. Each node in N_Q registers for newData, yielding notification if a new data object is inserted into the local N_Q partition. Subsequently, a *get* is issued on the other table. Matches are merged with the probe tuples and passed to the next stage of execution. Another join algorithm, *Fetch Matches*, is based on a standard distributed join algorithm which works on tables that already hashed the join attributes. Here N_R is scanned and for each R tuple a *get* is issued for the corresponding S tuple. Once S tuples arrive, predicates are matched and respective tuples are merged and results are

passed along as before. Especially the symmetric hash variant can consume a great deal of bandwidth. As optimisation, [66] proposes two join rewrite strategies. First, R and S are projected to their resourceIDs and join keys before the symmetric join is performed. Subsequently, the resulting tuples are pipelined into Fetch Matches join on each of the tables resourceIDs. Second, Bloom filters are created for each peer for its local S and R portion and published to a temporary namespace for each table. Filters are OR-ed together and multicasted to all peers storing the opposite table. Upon receiving a filter, peers begin to scan their corresponding fragment but limit rehashing only to Bloom filter matching tuples.

PIER provides an interesting approach to large-scale query processing. However, no mechanisms are provided to reflect the heterogeneity of peers. Similarly, it is not reasonable to assume that data namespaces are of uniform volume. PIER achieves scalability by relaxing consistency constraints. This limits application to scenarios where scalability is the main requirement. However, for certain tasks or sub-tasks that have no scalability issue but depend on consistent data management, PIER might not be well suited. PIER does not provide real-time capabilities.

2.4.3 Application Layer Multicast

The Internet was designed for one-to-one communication like E-mail and file transfer. Recently, applications like video-on-demand or live streaming and video conferencing have emerged that feature a one-to-many or many-to-many communication model. The IP multicast proposal [34] was aimed at providing global inter-network group communication but did not prevail due to the complexity of the design and limited understanding of commercial requirements [37]. For example, to implement IP Multicast specialised routers need to be installed at several levels of the network from backbone to edge routers. This constitutes considerable costs for Internet Service Providers (ISP).

The concept of Application Layer Multicast (ALM) implements the multicasting functionality at the application layer (OSI) using the unicasting functionality of the underlying IP network. The central benefit of the application layer solution is the straightforward and immediate deployment over large and heterogeneous networks. Although the approach is less efficient in comparison to, e.g. IP Multicast, the disadvantage is outweighed by the large-scale deployability, easier update and maintenance of the algorithms and adaptability to the user application. In ALM, nodes connect to an overlay to span a multicast tree. Links in the overlay are built using predefined metrics, e.g., delay or robustness. Node discovery and link stabilisation require additional bandwidth but are easily balanced by the advantages mentioned above.

Since the introduction of the ALM approach, a plethora of algorithms has been proposed. A categorisation based on the target application is presented in [37]. The categories are:

- Audio/video streaming: The distribution of audio and video content from a single source to a large number of receivers
- Audio/video conferencing: Real-time distribution of audio and video content in small groups
- Generic multicast service: A generic distribution service that can be parameterised through application domain specific metrics
- Reliable data broadcast and file sharing: Distribution of large files, usually in a distributed database or file sharing context. The metric is bandwidth.

Depending on the application domain, specific metrics apply. In the context of power networks, for instance, real-time distribution in smaller groups and configurable distribution services might be of relevance. Generally, ALM can be implemented directly on the end host, e.g. sender or receiver, or on an intermediate proxy overlay to which senders and receivers are connected.

An important concept of ALM approaches is the so called multicast group management. It includes discovery of multicast sessions, centralised or de-centralised administration and the mesh-first or tree-first approach to construct source specific or shared multicast trees. In a mesh-first approach, the overlay that links the nodes is known a priori. A routing algorithm, executed at a root node, determines the multicast tree. In contrast, the tree-first approach proceeds by building the tree without a pre-existing mesh. The algorithm is executed on each node, thereby providing the flexibility for local optimisation and stabilisation but requiring additional methods to detect loops and to ensure that the resulting graph is indeed a tree. Tree-first approaches might yield heavily unbalanced trees because upon reorganisation entire sub-trees are swapped without prior global optimisation. In this aspect, the mesh-first approach is superior as it is more robust and responsive to tree partitions. Hence, the mesh-first approach is more suitable for multi source applications. The question whether to administrate multicast groups in central or de-central fashion is application specific. While a de-central solution is certainly more scalable and more robust, it is also more complex to implement and stabilise. A central solution, on the other hand, is simple and easy to deploy but less reliable and scalable. Central solutions are therefore suited for small-scale applications while de-central solutions are superior in large-scale scenarios.

The key part of an ALM algorithm is the routing mechanism i.e. the, potentially heuristic, solution to a graph theoretic problem with consideration of certain constraints for each node. Four approaches to the routing mechanism are common: shortest path, minimum spanning tree, clustering structure and peer-to-peer structure.

Shortest Path

Aiming the construction of a degree constraint minimum diameter spanning tree, shortest path approaches use round trip times (RTT) to measure shortest paths between source and end hosts. Time delays are minimised and QoS parameters are taken care of. Among others SpreadIt [36] and TAG [82] employ the shortest path approach.

Minimum Spanning Tree

This approach ignores the degree constraint and constructs a minimum spanning tree (MST) i.e. a tree with minimal costs spanning all nodes of the network. MSTs are mostly used in centralised solutions such as ALMI [106] and HBM [120]. Both ALMI and HBM construct a low cost shared tree not routed at any particular source.

Clustering Structure

ALM protocols following a clustering approach to organise the network in a hierarchical cluster of nodes. Clusters are interlinked by dedicated cluster head nodes. Clustering the network to sub-groups reduces complexity but may yield sub-optimal solutions. Both ZIGZAG [144] and NICE [8] create cluster structures.

Peer-to-Peer Structure

In this approach the underlying P2P protocol is used for reverse or forward path forwarding. A prominent example using reverse-path forwarding is Bayeux [161]. Another example, Scribe [125], relies on forward-path forwarding while Borg [159] is based on both reverse and forward-path forwarding.

In summary, ALM provides several advantages over IP multicasting solutions. Among them, most important are: straight forward deployability, independence of the physical network and the option to optimise for specific application requirements. ALM protocols can be used in large-scale networks such as the Internet without prior modifications of the communication infrastructure. Some of the ALM concepts play an important role in other

related work, namely data-centric publish-subscribe systems, discussed in the following section 2.5.

2.5 Publish-Subscribe Systems

In a publish-subscribe system, information producers publish events to a global propagation mechanism often referred to as notification service. Subscribers state their interest in events by specifying filters for specific events. Events are not directly addressed to subscribers but sent asynchronously through the notification service.

The notification service acts as de-coupling entity between data producers and consumers. The interaction between producers and consumers is de-coupled in respect to three aspects: space, time and synchronisation. Producers are not aware of whom they send events to and, similarly, consumers do not know who sent the events they receive[3]. They do not need to participate in the interaction at the same time and publishers are not blocked when producing events as well as consumers are notified asynchronously of new events. To implement this features, notification services need to provide the following functional components (i) management of subscriptions for all subscribers, (ii) reception of publications from publishers, (iii) routing of events to subscribers. Generally three type of architectures exist for notification services. They can be fully centralised with consumers and producers sending messages to a single entity which stores and forwards them accordingly. Alternatively, messages can be exchanged directly between producer and consumer without an intermediate entity. Finally hybrid architectures exists where the notification service is implemented as network of servers.

Notification services further differentiate by the subscription model they support. In the following, different subscription models and prominent implementations are discussed.

2.5.1 Topic Based Subscriptions

In this model, an event notification is grouped by a topic T. Topics can be structured into hierarchies. Subscribing to a topic T will the subscriber cause to receive all events tagged as T or by any of the sub-topics $T_{s0}...T_{sn}$. Systems utilising this model can be implemented efficiently due to static routing. However, due to the limited expressiveness, subscribers may receive events which they are actually not interested in. Systems implementing topic based subscriptions are SCRIBE [125] which employs Pasty [124] for

[3]Of course internal event semantics may determine information in order to identify the source of the event.

event dissemination. In this multi-broker architecture subscriptions are routed along the Pastry routing tree towards a broker server which is responsible for the management of that particular subscription. Similarly, events are disseminated by travelling the routing tree to the corresponding broker which selects the respective multicast tree to forward the event to the clients. Other examples supporting topic based subscriptions are Bayeux [161] which employs Tapestry [160] for event dissemination.

2.5.2 Content Based Subscriptions

Content based subscriptions allow the subscriber to define a filter by specifying several criteria. Filters are usually formulated in a subscription language. Subscribers receive all events that match the criteria provided in the subscription. This model provides more flexibility as criteria can be specified at runtime. As a disadvantage, the approach yields higher runtime overhead as filters require considerably more processing time. Examples of systems proving content based subscriptions are Siena [20], Evlin [128] and Gryphon [140] where subscribers can specify an event type and a set of predicates using SQL 92 syntax.

2.5.3 Type Based Subscriptions

Type based subscriptions are similar to topic based subscriptions. Instead of specifying a filter that matches the event content, subscribers specify the *type* of subscription they are interested in. The subscription may include a set of predicates which further filter events according to the subscribers' interest in the event content. Types can be structured in hierarchies causing the subscriber of a super-type to receive all events of sub-types that match the specified predicate set. This approach combines the simplicity of the topic based model with the expressiveness of content based subscriptions. Type based subscriptions are supported for example by Hermes [108].

2.5.4 Quality of Service

Notification services provide a variety of qualities of service. Most common are persistence, transactional guarantees and priorities. A service provides persistence by ensuring that published messages are not lost in the messaging system, even in the event of system failure. With transactional features, sequences of events are guaranteed to be received by subscribers either in full or not at all. Messages can have assigned priorities which influence the transit through the notification service. Messages with higher priorities, for

example required for real-time applications, are routed first while message with lower priorities may be delayed.

The various flavours of publish-subscribe systems enable flexible routing of information from multiple sources and sinks. Topics and types allow to make routing decisions based on application requirements and event content. Using query languages and predicate sets to control routing is also exploited in the related research field called *declarative routing* which is elaborated in the next section.

2.6 Declarative Networking

The term *declarative networking* was originally coined by the research team around Boon Thau Loo from UC Berkeley [90], [92], [131], [91]. It represents an approach to design and implement distributed protocols and algorithms by declarative specification as distributed and recursive queries over network graphs. In this context, distributed queries adjust and maintain routing tables of nodes recursively over arbitrary long multi-hop paths of a network [92]. In [91], the compactness and flexibility of the declarative specification is demonstrated by providing a complete implementation of the Chord protocol [138] in just 47 lines of OverLog rules.

The concept models the routing infrastructure as a directed graph. Each link has associated a set of parameters such as loss rate or bandwidth. Nodes can be either IP routers or overlay nodes (peers). The routing scheme is fully distributed with each node being equipped with a general purpose query processor. Additionally, each node maintains links to its neighbours (neighbour table) and forwarding information to route packets (forwarding table). The query processor updates the forwarding table either periodically or upon notification. Upon receiving a request, the query processor may initiate further distributed execution in the network. Execution results can be either used to update the forwarding table, or sent back to the issuer where it can be used for further processing.

In [90] Loo introduces network datalog *NDlog*, a subset of the Datalog [112] language for declarative network specification. Datalog programs consist of a set of declarative rules and a query. The query specifies the requested output. A rule has the form $p :- q_1, q_2 ... q_n$. p is the head of the of rule and $q_1, q_2 ... q_n$ a set of literals that can be either predicates or functions applied to fields. The set of literals constitutes the body of the rule. The commas separating the predicates are conjuncts. Recursion can be expressed by referring to each other in a cyclic fashion. Listing 2.1 shows a Datalog example program which computes the set of all paths based on input link tuples. In the listing S,D,C and P stand for source, destination, cost and pathVector fields.

Listing 2.1: Datalog Example

```
NR1: path(S,D,P,C) :- link(S,D,C),
    P = f_concatPath(link(S,D,C), nil).
NR2: path(S,D,P,C) :- link(S,Z,C1),path(Z,D,P2,C2),
    C = C1 + C2,
    P = f_concatPath(link(S,Z,C1),P2),
    f_inPath(P, S)=false.
Query: path(S,D,P,C)
```

Rule NR1 generates one-hop paths and stores them at the source node S. Rule NR2 recursively generates paths by matching destination fields of existing links to source fields of earlier computed paths. In other words, if there is a link from S to Z and there exists a path from Z to D then there is a path from S to D via Z. The function `f_inPath(P,S)` returns `true` if the source node S is part of the path P, hence the generation of cyclic path is prevented.

NDlog extends Datalog by providing explicit control on data placement and transfer. It is accomplished with a location specifier written as the first field in all predicates, e.g. for `link(@S,@D,C)` the location specifier is `@S`. NDLog does not assume that all nodes in the network are directly connected, but rather that nodes are connected to a comparatively small set of neighbours. To state that two particular nodes are connected, the link relation `link(@src, @dst, ..)` is used. The first two fields indicate source and destination addresses of the nodes followed by an arbitrary number of fields describing metrics or other features of the link. Since query execution is distributed, NDlog provides the concept of local rules to indicate that a specific query does not need communication. Local rules are simply rules where predicates have the same location specifiers. The explicit communication along physical links is expressed by a link literal in the body of a rule marked by `#`. Finally, to restrict communication to physical links, the link restrict rule is defined as either a local rule or a rule that has exactly one link literal in the body and all other literals have their location specifier set to either the source or the destination field of the link literal. A NDlog program can be defined as a Datalog program where (i) each predicate has a location specifier, (ii) an address variable cannot appear as other typed variable in the rule, (iii) link relations never appear in the head of a rule with an non-empty body and (iv) any non-local rules are link restricted by some link relation.

Listing 2.2: "Shortest Path in NDLog"

```
SP1: path(@S,@D,@D,P,C) :- #link(@S,@D,C),
    P = f_concatPath(link(@S,@D,C), nil).
SP2: path(@S,@D,@Z,P,C) :- #link(@S,@Z,C1),
    path(@Z,@D,@Z2,P2,C2),
    C = C1 + C2,
    P = f_concatPath(link(@S,@Z,C1),P2).
SP3: spCost(@S,@D, min<C>) :- path(@S,@D,@Z,P,C).
SP4: shortestPath(@S,@D,P,C) :- spCost(@S,@D, C),
    path(@S,@D,Z, P,C).
Query: shortestPath(@S,@D,P,C).
```

Listing 2.2 depicts an NDlog program calculating shortest paths between nodes. SP1 generates one-hop link tuples while SP2 generates multi-hop paths between nodes. SP3 derives the relation `spCost(src,dst,mincost)` that computes the minimum cost for each input path. The angle bracket notation specifies the minimum aggregate construct. SP4 derives the shortest paths with cost and path input. Eventually, `Query` specifies the shortest path tuples as result.

P2 [92] is a framework for the declarative construction of overlay networks. Applications submit logical descriptions of the overlay algorithm which P2 compiles to executable function to maintain routing tables, perform resource discovery and provide forwarding for the overlay. P2 differentiates from other overlay construction frameworks in that it features a declarative logic language to specify overlays and that it utilises a data-flow framework to maintain the overlay instead of the traditional protocol state machines. P2 programs are compiled into a data-flow representation and deployed on network nodes where they execute. In a data-flow graph a variety of database operators are connected through edges which represent the flow of tuples among operators. The P2 query language called OverLog allows to express overlay networks in a highly compact and reusable form. For instance, the Chord protocol has been implemented with just 47 lines of code compared to the thousands of lines of the original implementation. The high level overlay description, however, comes at the price of reduced performance. Optimised C, C++, and Java implementations can perform considerably better using host resources more efficiently. Therefore P2 is generally aimed at rapid prototyping scenarios and less at production systems.

In P2, overlays are modelled as distributed data structure represented as structured relations similar to relational databases. Two types of tuples are supported: soft-state tables or streams of tuples. A relational model is beneficial first, because network state can be intuitively expressed by structured tables and, second, tables and relationships can be

expressed concisely in a declarative query language. OverLog is based on Datalog with extensions to specify physical distribution properties, continuous queries over streams as well as tables and deletion of tuples from tables.

The P2 runtime provides the basic classes `Tuple` and `Value` to represent data in the system. A Tuple is a vector of Value objects. Tuples, Values and operators are translated into an intermediate language called PEL which in turn is compiled to byte-code. A virtual machine executes the resulting byte-code. Execution is single threaded and event driven requiring blocking and long running events to be assigned to additional threads.

Tables in P2 are queues of limited size and limited validity for individual queue elements. Tables are referenced by unique IDs and are visible to all queries currently executing. They are local data structures but location specifiers in the OverLog rules allow transparent partition of data over several nodes. True predicate indexes allow efficicent lookup of tuples.

2.7 Summary

This chapter introduced current technologies for large-scale data-centric systems. Starting with a general introduction in architectural methods, it became clear that large-scale systems pose new challenges that require methods beyond the traditional architecture business cycle. Open source models as suggested as part of the ULS research together with Web 2.0 platform approaches constitute methods to design, maintain and evolve platforms for large-scale industrial infrastructures. To implement such models, corresponding software architectures and platforms are required.

Databases are today the number one choice for data-centric integration. Strictly assuring atomicity, consistency, integrity and durability they hide storage complexity from applications. The rich feature set, however, limits their applicability in the face of increasing data volumes and update intensive load patterns. Alternatives range from distributed file systems like GFS to new data models like Bigtable and Hbase. While databases or database clusters can be classified as distributed systems with a relatively low degree of distribution, P2P systems lie on the other end of the spectrum. Being designed to co-ordinate collaborative resource use of millions of peers, P2P systems enable data management for very large infrastructures. Due to their self-organising capabilities, maintenance overhead remains at a minimum. However, the ability to scale to large systems comes at the price of relaxed transactional behaviour and no support for queries beyond keyword searches.

Besides querying, a key challenge is the bi-directional dissemination and collection of

information among networked nodes. Building upon the addressing schemes of overlay networks, information collection and dissemination methods can be implemented. This virtualisation from physical networks, reduces the complexity for the application developer.

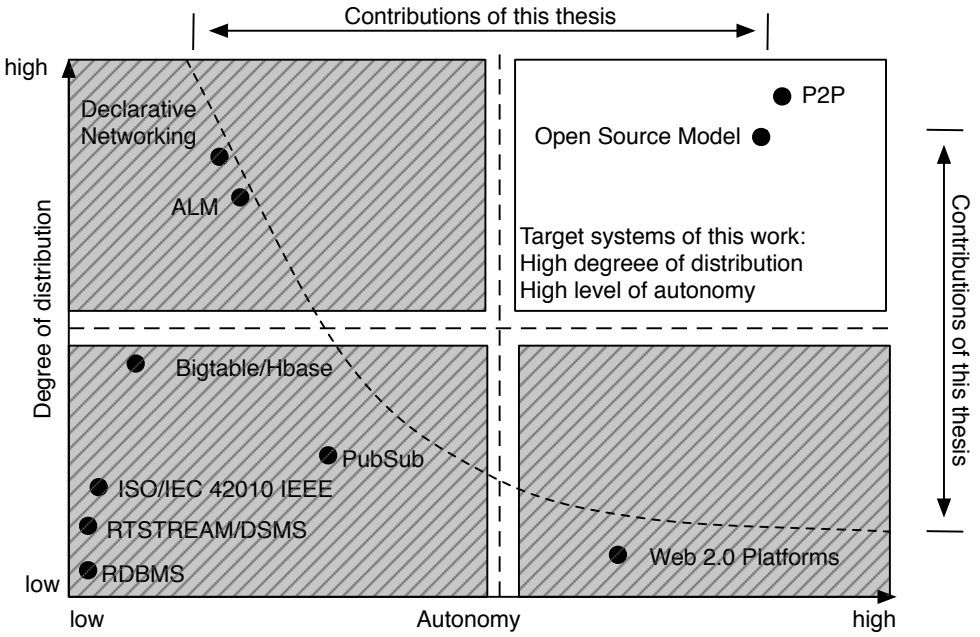


Figure 2.4: Technology cluster by degree of distribution and autonomy

Recalling the research questions and challenges of the target systems as elaborated in Section 1.5, two essential characteristics need to be fulfilled by technologies in order to achieve quality attributes in large-scale systems. The first is *distributed organisation* determined by the inherent structure of the system, which is composed of individual networked parts that are globally distributed and do not synchronise with a central point of control. The second characteristics is *autonomy* or *reverse administrative proximity* [46]. As individual parts are owned, maintained and operated by different parties, no single entity has complete access to all of the parts nor can the behaviour be reliably predicted. Figure 2.4 clusters the technologies and methods reviewed in this chapter along the two dimensions.

Although RDBMS may run on clusters their level of distribution is small compared to large systems with thousands of entities. Some DSMS exhibit a slightly higher level of distribution due to in-networking aggregation and pre-processing. Traditional architecture and design methods are largely centralised with an architect co-ordinating the whole project. Bigtable and Hbase are large distributed systems, yet the level of de-central organisation is low due to their flat hierarchical organisation. Publish-subscribe systems realise loosely coupled distributed systems, hence they allow individual parts to operate autonomously. Although realised on server farms, the degree of de-central organisation is

low. While web 2.0 platforms may technically be distributed, i.e. to multiple servers, they are operated by a single party and hence appear as a central entity. They do, however, allow high levels of autonomy as users have direct influence on the service and can, technically, enhance the service by supplying software that runs in the platform. Due to its recursive execution, declarative routing rules such as those specified in OverLog are highly distributed. Autonomy, however, is low since routers are most likely maintained by a single party. Similarly, ALM systems which can span large hierarchies yet strictly control each participating node. Both P2P systems as well as the open source model are located at the extreme of both dimensions. Both allow a maximum of autonomy for the individual yet do not have a determined central point of control. The level of de-central organisation of the open source model is slightly lower than for P2P systems since a single or team of democratically elected leaders are able to make global decisions. The systems under investigation in this work reside on the upper right quadrant of Figure 2.4. Autonomy comes from the large number of individual components that function together. The degree to distribution is inherent in the system due to the physical separation of entities as well as the heterogeneity of entities. The open source model and P2P technologies seem good candidates to address the challenges for large-scale systems. Yet both emerged from domains other than industrial, i.e. IT or media, and therefore do not meet the requirements presented in Section 1.3. Therefore, in order to address the research questions stated in Section 1.5 this work contributes by the adaptation and transition of the these technologies into the target domain of large-scale industrial systems.

The following chapter will introduce specific background on power infrastructures. In a section on the Smart Grid the above illustration will be complemented with a classification of the Smart Grid into the two dimensions. The Smart Grid is representative for a large-scale industrial system in the power domain.

Chapter 3

Power System Infrastructures

Consisting of thousands of sensors and actuators, power systems are among the largest and most complex technical systems man has ever made. This chapter introduces power system essentials both from an electric and from an information and communication infrastructure perspective. By illustrating power generation, transmission, distribution and usage, the influences of different load types are described. Moreover, the need for control and monitoring infrastructures as well as challenges for future architectures are motivated. It remains important to note that this introduction is neither comprehensive nor complete but meant to provide a general overview and introduction of key terms and concepts.

3.1 Power Systems Essentials

Power systems constitute the backbone of modern society (Figure 3.1). Electricity is regarded as commodity with almost constant availability. Without electricity rail systems such as subways and trains would come to standstill, traffic lights would not function, computers would not work, water supplies would stop or run out. In short: modern society would collapse without electricity.

The high level of availability, is even more remarkable when considering that what lies beyond the AC outlet, is a highly dynamic and complex system. Large power systems exhibit a variety of dynamic phenomena regulated by various types of controllers. Starting from simple on/off switches like circuit breakers to isolate short circuited or malfunctioning equipment, the range spans over discrete controllers like tap-changers in transformers to continuous controllers like voltage controllers and power electronic controls in Flexible AC Transmission System (FACTS) devices which can control power flow or voltage.

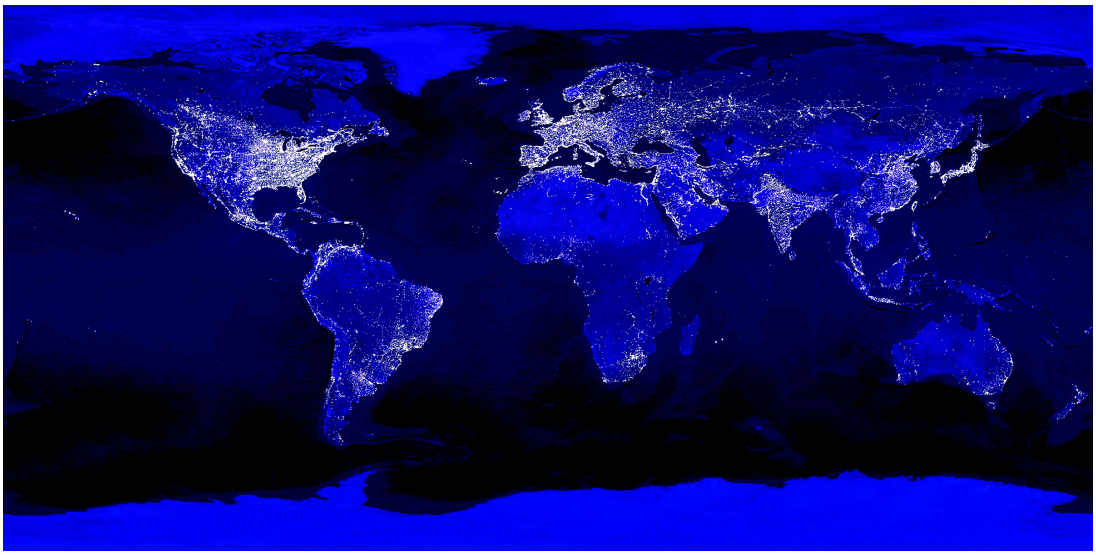


Figure 3.1: Earth Lights - Data courtesy Marc Imhoff of NASA GSFC and Christopher Elvidge of NOAA NGDC. Image by Craig Mayhew and Robert Simmon, NASA GSFC

Most controllers act locally, e.g. protection systems operating on measurements within the same substation. Most dynamic characteristics, however, emerge as regional or even system-wide patterns. Hence, power system controllers need to control the global system state via local actions. Before proceeding with an overview of stability and control approaches, the following sections introduce a common taxonomy for the power system domain.

3.1.1 Power System Key Concepts and Components

In power systems, a small number of generators generate electric energy which is then transported over the transmission and distribution network to a large number of consumers. Following the hierarchical organisation inherent in today's power infrastructures, the following paragraphs introduce the key elements of a power system from generators, over the transmission and distribution network to consumers which are also, more generally, referred to as loads.

Generation

Generation describes the process of transforming natural energy resources to electrical power. Natural energy, e.g. potential energy of water, energy derived from combustion or atomic reactions etc., is conveyed to turbines where mechanical energy is transferred to alternators which accomplish the actual transformation to electrical power.

A generator requires several monitoring and control equipment, i.e. the automation and process control system, before it can function in a grid. The automation system executes regulation instructions from the Energy Management System (EMS) in the control centre of the utility. Both the generator and its turbine are subject to the regulation process. Control commands include adjustment of the turbine torque which determines the current from the generator. The control of the spin speed determines the frequency and the control of the current in the exciter coils of the alternator which determine the amplitude of the output voltage.

Recently, traditional power systems, i.e. systems without generation at the distribution level, started to evolve towards systems having production at two levels. One level is constituted by already existing large-scale power plants that are connected to the high voltage transmission network. Another level is composed of a large number of Distributed Energy Resources (DER) connected to the low and medium voltage distribution network. However, due to the lack of detailed real-time information, in traditional power systems, certain areas of the transmission and distribution network appear as black boxes to network operators. In a system with a well-defined top-down power flow detailed real-time information is not necessary since network states can be estimated with acceptable precision. In the new network architectures, however, power flow is unknown for large areas because distributed generators may feed more into the network than they consume. As a side effect, protection systems, unaware to the possibility of bidirectional power flow, may fail thereby endangering human life and causing equipment damage. Substantial investments are required before distribution networks are able to cope with the integration of a large number of distributed units. On the technical side, key challenges relate to the control and co-ordination of the large number of small DER, advanced protection systems, network reconfiguration and power flow control.

In recent years, a new market for so called *micro-generators* emerged. This special form of DERs target domestic markets and produce electrical energy from a large spectrum of (waste-) energy sources. Most common and highly developed are:

- Gas combustion turbines
- Gas combustion microturbines
- Gas to hydrogen fed fuel cells
- Wind powered generators
- Photovoltaic cells
- Solar thermal-electric power plants

- Hydroelectric micro- and small-scale power plants
- Geo-thermal steam turbines

Additionally, batteries, fly-wheels, super-capacitators and other types of energy storage are regarded as DER. Due to their ability to absorb peak loads and balance consumption and production, energy storage systems will play an important role in the context of integration of stochastic generators such as wind and solar into the global power grid.

DER are potentially operated by end users which may switch their role from consumer to producer several times a day. End users aim to maximise their benefit regarding the operation of the DER. For instance, the operator of a bio-gas turbine may connect only if prices reach a certain threshold and disconnect abruptly if the price drops below a certain limit. Similarly, close-by wind turbines may start feeding enormous amounts of electrical energy as winds kicks in and photovoltaic systems will drop output as clouds disguise the sun. Energy Management Systems (EMS) are required to cope with the dynamics and complexity induced by such unforeseeable events.

Transmission and Distribution Network

The *Transmission and Distribution (T&D) network* transports electric energy from generator to consumer. The transport is accomplished by overhead and underground lines. The distance between generator and consumer determines the fundamental design of the transmission and distributed network. The larger the distance and the higher the amount of power to be transported, the higher the system voltage. The transmission network connects power plants with transmission *substations*. It usually spans the largest distances, hence voltage in transmission networks is the highest. Substations transform voltage and supply the *distribution network* which distributes energy to connected consumers.

The number of voltage transformations from highest to lowest voltage level determines the network topology. In a *radial* topology all substations are fed by a single supply. Radial networks are less expensive to built but are also less reliable. A *loop* topology connects each substation with two supplies or at least with one supply from two directions. Loop topologies are more reliable but also more expensive. In a *multi-loop* topology substations are fed from more than two directions and hence are even more reliable but also more costly.

Connecting power carriers, i.e. overhead lines and underground cables, substations are the nodes of the (T&D) network and hence play a key role in the control of a power system. Incoming and outgoing carriers are connected to so called *busses* or *busbars* by *feeders*, i.e.

circuit breakers, *disconnectors*, and *instrument transformers*. Transformers in substations interconnect the different voltage levels. Substations further host the *protection systems* and transmit measurement signals to the control centre.

Protection systems are designed to clear faults such as short circuits which can damage busbars, transformers or lines. The operation principle is rather simple. Transformers provide measured values of the current voltage and current levels to a protective relay. The relay applies its protection algorithms to determine whether to operate a specified circuit breaker hence isolating faulted sections or equipment. Modern relays are computerised featuring communication facilities, self diagnosis and event recording. At a conceptual level protection methods are trivial. However, isolating a faulted segment while leaving healthy segments in operation is a complex undertaking. Protection must be sensitive enough to react quickly but also provide stability and continuity during operation close to the capacity limit of feeders, lines and cables. Further complicating are anomalies like lightning strokes which constitute a rather transient phenomenon which should not yield an interruption of supply. Typically, transient faults are compensated with a re-closing mechanism, i.e. the relay closes the circuit thereby checking whether the fault is still present. If this process failed several times the fault may not be able to be cleared locally.

Protective relaying systems are usually built with various levels of redundancy to isolate fault conditions and equipment quickly as well as maintaining stable system operation. While local backup systems are easily disabled by severe component failures, *remote backup systems* provide additional security by physical separation.

Transformers

Transformers are essential components of the AC power system as they enable conversion between different voltage levels with high efficiency. Power is generated at relatively low voltage between 10/25kV, then, to reduce losses during transmission, it is transformed to higher voltages between 110kV-420kV before it is transformed down to 400V for domestic consumption. Conceptually, transformers consist of two coils around a common iron core implementing a magnetic coupling between the coils. Considering an ideal transformer, i.e. no resistance, no leakage flux and infinite permeability of the core, the relation between the voltage on the primary side and the induced voltage on the secondary side can be written as:

$$\frac{v_1}{v_2} = \frac{N_1}{N_2} = n \tag{3.1}$$

Where v_1 is the applied and v_2 the induced voltage, N_1 the number of turns of the primary transformer winding and N_2 the number of turns of the secondary winding respectively. So called tap-changing transformers are able to control the number of turns and hence

can be used for voltage control. In the non ideal case, transformers have several dynamic characteristics that can influence Power Quality (PQ), i.e. constant voltage and frequency, and and general power system stability. When the transformer core has been disconnected from the grid, it contains a residential magnetic flux ϕ_r. Upon reconnection, the grid voltage initiates a flux in the same direction hence the total flux becomes $\phi_r + \phi_s$. The core material goes into saturation which causes large current inflow from the system. This current inrush can take several seconds to disappear.

Loads and Consumers

Loads usually convert AC electrical energy into other forms of energy, e.g. mechanic, light, heat, DC electrical energy and chemical energy. Typically, loads are categorised into *residential*, i.e. domestic users, *industrial*, i.e. commercial users, and railways. In the following, different load types (motors, light bulbs, AC/DC converters, batteries) are discussed. Moreover, the effects, induced on the power grid by each of the load types, are explained.

Transformation from electric to mechanical energy is done by motors. Most motors are AC machines, i.e. synchronous motors and asynchronous motors. DC motors are utilised by trains and as drivers in hard-disks which benefit from their traction properties. For AC motors the three-phase power supply generates a rotating field which acts as torque on the rotor. Synchronous machines can act as generators and as motors. They operate at constant speed unless supplied with power-eletronic converters to control the frequency of the power supply. Around 60% of the total supplied electrical energy is consumed by asynchronous motors which are 90% of all electric motors in use. They operate in dishwashers, washing machines and air conditioners. Asynchronous motors are not synchronised with the rotating field. This influences the power system when the machine starts, stops or the mechanical load changes. The motor is driven by applying the three-phase voltage on the terminals of the strator windings producing a rotating field which in turn induces a rotating current in the windings. The field plus rotating current generate the torque on the rotor. During startup the applied current is much higher than nominal. Rotor current decays as spinning speed increases. The startup phase of large asynchronous motors has the same effect on power systems as the inrush current of transformers. In steady state, the motor develops a torque equal to the mechanical load. However, rotor speed must always be sufficiently less than synchronous speed in order to develop the torque needed to balance the mechanical torque. The difference in velocities is referred to as *slip*. If, under heavy load conditions, supply voltage decreases, the electromagnetic torque decreases as well causing the slip to increase which in turn increases the current in the windings.

Hence, an asynchronous motor amplifies supply voltage drops and, in the worst case, may cause blackouts or complicate the operation of reconnecting feeders.

In traditional light bulbs electric current heats a thin filament which in turn radiates light. Dimmers used to control the brightness of the bulb chop the applied AC voltage and current accordingly. The application of dimmers yields non-sinusoidal and harmonic currents in the grid. In the presence of many harmonic current inducing loads, voltage may become distorted as well as yield decreased power quality.

Home appliances like heaters and water cookers convert electrical energy into heat through an resistor. The aluminium and steel-making industry applies electric heating at a large-scale. A steel-making furnace induces similar dynamics as a transformer on the network. The melted metal short circuits the secondary winding and the current heats up the furnace.

Most electronic equipment requires DC electrical energy and hence rely on AC/DC conversion. AC/DC converters are built from diodes which act as switches conducting the current across only if the applied voltage is positive.

Batteries directly convert electrical energy into chemical energy when charging and vice versa when supplying. Having positive and negative terminals immersed in a solid or fluid electrolyte, electric current separates protons and electrons thereby creating an electric potential between the terminals. On discharge the potential between the terminals levels out.

Despite the dynamics induced by the different load types just introduced, grid stability and PQ must be actively maintained. The following section introduces control values, methods and architectures which are employed in state of the art control facilities.

3.2 Power System Control

Power system loads and consumers are supplied with power at near constant frequency and voltage. Since electrical energy cannot be stored efficiently, the balance between generation and consumption must be actively maintained by control actions.

Commonly it is distinguished between primary, secondary and tertiary control. Primary control occurs at a millisecond scale, e.g. excitation control on generators to regulate voltage. Secondary control has time windows of up to 15 minutes and includes, for example, the Transmission System Operator's (TSO) actions to balance a specific control area. Tertiary control has a time horizon of up to 24 hours and includes transmission schedules and consumption as well as weather forecasts. Besides temporal characteristics,

control actions can be classified according to their controlled values, i.e. *voltage control*, *power flow control* and *frequency control*. The following details each of these control types together with their scope of control, i.e. local and system-wide.

As part of primary control, voltage can be controlled by adjusting the excitation current in a generator. Additionally, voltage can be controlled by utilising tap changing transformers, shunt capacitors, or reactances which are operated as part of an automatic feedback loop.

While tap changes and shunt reactances are discrete controllers, more recent power electronic devices such as Static Variable Controllers (SVC) allow for more continuous and faster control. Voltage control happens at a local scale. However, control actions have influence on other parts of the network as well.

While power injections and voltages are controlled precisely, power flows at transmission lines are usually not controlled. However, so called phase shifting transformers are capable of controlling the power flow. The control is discrete, slow and local. The flow over DC lines is always controlled and control is very fast.

Generally frequency is controlled by balancing load and generation. Sensing deviations in speed at the generator, mechanical input power is adjusted to achieve constant frequencies. Primary frequency control is local and very fast. Secondary control, also known as Automatic Generation Control (AGC) and Load Frequency Control (LFC) is done at the control centre, hence not local and slower.

3.2.1 Control Centres

Control centres collect information on the current state of the power system. Based on the data acquired, the operator can monitor the system and take corrective action if necessary. The set of control actions an operator can manually initiate include opening and closing circuit breakers and changing transformer taps. Data acquisition and operator controls are subsumed in so called Supervisory Control And Data Acquisition (SCADA) systems.

SCADA systems retrieve field data from Remote Terminal Units (RTU) which are installed in substations as well as power plants. RTUs can communicate over several communication infrastructures such as optical network or telephone lines. Transmitted data includes switch status (On/Off) of circuit breakers as well as voltage and power measurements.

While voltage control and protection belong to the local controls, frequency control is the only wide area control task implemented in control centres. Frequency control is achieved in a feedback control loop starting with the capture of measurements on generator outputs and tie-line flows. Based on the collected data, generator governor set-points are computed.

The accuracy of the calculation is restricted to the data collection interval which is between 2-4 seconds for both retrieval and control commands respectively.

With the availability of more powerful digital computers, control centres began to host additional applications. Most notable is the *state estimator* which calculates a real-time steady state model of the power system. Based on the model, disturbances can be analysed giving the operator a timely chance for corrective action. Furthermore, the model is used to analyse the current state of the system aiming to identify alternative, more optimal configurations. SCADA systems that are extended by these new applications are also referred to as Energy Management Systems (EMS).

The rapid evolution of digital computers led also to a new generation of substation equipment. Being able to sample data at a millisecond rate, devices are capable to compute highly detailed state assessments. However, due to the limitation of the communication infrastructure, this high volume real-time data is not transmitted to the control centre but stored locally in limited quantity. Therefore its use is limited to off-line studies or post-mortem analysis.

3.2.2 Control Architectures for Distributed Generation

Increasing integration of distributed energy resources in the distribution system structure gives rise to various problems. Among them are mis-operation of protection equipment, poor power quality, inadequate voltage profiles and stability problems. A concept called *active distribution network* aims to address these issues and support large-scale penetration of DER in distribution systems [28]. Several research groups investigate active network concepts providing architectures, control paradigms and strategies for integration into the standard grid. The latter is mandatory as massive re-design of existing distribution networks is not feasible. Therefore, active network proposals keep the existing infrastructure, e.g. protection and control systems in place, and rather extend them with a new control functionality, e.g power flow control.

Voltage and frequency control of current power networks assume that all or most of the generated power comes from a few large generators at the transmission level. In the future, control mechanisms need to be tailored for the distribution level. Hence, at this level, a new set of control functionality is required, namely: balance production and consumption, maintain frequency and voltage levels, control of PQ, measure and synchronise connect and disconnect of isolated networks with the main grid. In the following paragraphs, several architectures to implement this control functionality are introduced.

Eltra [102] initiated the Cell Controller Project which introduces the metaphor of a distri-

bution cell, similar to a broadcast cell in a mobile network, to describe 60kV sub-networks below 150/60kV transformers. In emergency situations, a cell is disconnected from the High Voltage (HV) grid and transferred to controlled island operation. A cell provides explicit support for:

- Online monitoring of loads and production
- Active power control of generators
- Capability of remote breaker operation
- Voltage and frequency control
- Black start support to the transmission grid

Besides the concept of cells, another approach gained widespread popularity in recent years. It is motivated as follows: a considerable source of inefficiency in generation is the fact, that, in order to generate electricity, heat is generated which is then transformed into electricity. Subsequent to its generation, power is transported to the consumer, who often uses electricity to generate heat again. The next section introduces the concept of *microgrids*, which aim for high efficiency by combining local heat and electricity generation.

Microgrids

The *Microgrid* concept introduced by the Consortium for Electric Reliability Technology Solutions (CERTS) [84], [107] combines load and microsources to operate as a unified structure for both electric power and heat. A microgrid is a small, isolated section of the main grid which can be operated independently. Similarly to the cell concept, microgrids provide distribution level control functionality, i.e:

- Voltage control
- Power flow control
- Load sharing
- Functionality for the smooth connection and disconnection to and from the main grid

Microgrids constitute a novel network structure located down stream at the low voltage (LV) layer [85]. In a microgrid, DERs such as microturbines, fuel cells, photovoltaic arrays

and distributed storage (DS) as well as controllable loads, e.g. air conditioners, are locally networked such that they can be controlled independently from the main grid. Moreover, microgrids are connected with the medium voltage distribution network to ensure power quality and stability but provide the option of isolation from the main grid in the event of failure [84]. For the consumer, microgrids provide benefits in terms of reliability, sustainability, improved power quality and decreased costs. In the context of the utility, microgrids are beneficial due to decreased transmission facility usage, increased service quality and better utilisation of transmission and distribution networks.

Control and management of microgrids differ substantially from conventional power systems. This is due as follows:

- steady-state and dynamic characteristics of DER units are different from large turbine units
- microgrids are subject to significant degree of imbalance due to single-phase loads and DER units
- certain sources in a microgrid have stochastic behaviour, e.g. wind turbines, photovoltaic arrays
- energy storage units can play an important role in microgrids
- microgrids must cope with constant join and leave of DER units, e.g., for economic reasons
- in addition to electrical energy, microgrids are also an important producer of thermal energy in form of waste heat

Microgrid control systems may be based on a central controller or embedded in each distributed generator or other equipment. In the central case, the microgrid central controller (MGCC) controls the actions of all components of the microgrid thereby optimising its utilisation [59]. When isolated from the main grid, the control system must operate the local control functions.In this control scenario, frequency control is particularly challenging. In conventional systems, frequency response is based on rotating masses. Since micro turbines, fuel cells and photovoltaic arrays are basically inertia-less and have slow response or ramp times, the behaviour of directly connected rotating masses must be imitated co-operatively by the electronic converters [93]. Voltage regulation, on the other hand, is a local problem and hence similar in connected and isolated mode. If the microgrid was exporting or importing power from the main grid before isolation, generation and consumption needs to be balanced. If demand exceeds current supply, demand side

management, e.g. load shedding, are implemented to maintain stability. By co-ordination of storage units additional stability is achieved in case of abrupt fluctuation of generation or consumption. Storage units are also utilised to maintain power quality by injecting or absorbing real or reactive power. When grid-connected, microgrid control functions are reduced to satisfying all of its load requirements and contractual obligations with the main grid.

Especially in isolation mode, communication between components is of utter importance. The underlying communication infrastructure must be of low latency and highly reliable. In the context of microgrid control, intelligent agents are frequently suggested as an enabling technology [117]. In this context, the different modes of co-operation between DER and DS are key to maintain integrity of the isolated microgrid. The following section elaborates on the importance of the communication infrastructure for local and system-wide operations.

Communication and Control System

Previous sections stressed already the importance of advanced information and communication technologies as an enabler for future power systems. Already today automation equipment provides standard networking via ethernet and TCP/IP. Importantly, the extra costs for digital equipment and sensors are quickly consolidated through improved operation and maintenance efficiency. Communication in an energy automation system includes functions for (electrical-) network reconfiguration, voltage frequency control, generation control, load control, control of active compensation devices, real-time monitoring, predication of consumption, generation and pricing. Control systems can be organised in various topologies: using centralised controllers, complete de-centralised with direct peer-to-peer communication or hybrid variants of the previous two approaches. Controllable equipment can also be autonomous, e.g. utilising local information when responding to grid events. This approach is particularly appealing when extremely fast responses are required.

Information and communication technology is essential for acquiring, storing, processing and distribution of information in power systems. As elaborated in [127], communication infrastructures might include different media such as landline, wireless or power line carrier. However, each technology might be more or less suited in a particular scenario: while landline is well suited for high bandwidth demanding applications, wireless brings increased flexibility but is limited due to sensitivity for magnetic distortion [127]. Power line is often used for automatic meter readings (AMR) but is not applicable for reclosers, switches and sectionalisers, as communication is lost on open circuits.

For the stable operation of an active distribution network, accurate knowledge of the network condition is a precondition. The network condition consists of network topology, properties and condition of equipment and information on voltages and current flows. As a common concept, monitoring data is processed close to where it has been sampled. Modern energy automation equipment, frequently referred to as Intelligent Electronic Device (IED), provide already functionality to process sampled data. From the IED, data is passed to a data concentrator, for instance a substation control unit, where it is further aggregated and eventually sent to the network control station.

Instead of this central control approach, the benefits of P2P communication have been investigated in the context of distribution protection systems in [35]. In traditional recloser protection schemes enormous pressure is put on reclosers due to high thermal and mechanical forces. Moreover, the voltage drop might cause power quality to decrease. Using P2P technology, devices are enabled to directly exchange their current status. This facilitates fast fault location and saves further reclosing operations. Relays locate faults by comparing measurements of all relays in the protection system. For instance, if a relay measures the fault current but its down stream neighbour does not, the fault is located between them. The device located closest to the fault initiates the reconfiguration process by sending open-close-lock commands to all respective relays in the protection system.

Regardless of concrete technologies employed, communication, objects models and protocols must be standardised in order to achieve exhaustive coverage. To cope with the complexity introduced by the new digital equipment, concepts to keep engineering efforts at minimum are required. Self-configuring systems and plug and play paradigms will play an important role in order to achieve economically feasible solutions. These advanced communication paradigms constitute an enabler for a new kind of power infrastructure labelled as the *Smart Grid* which is described in the following section.

3.3 The Smart Grid

Recently, the term *"Smart Grid"* has become the new and dominant buzzword in the power industry. The term is not defined precisely, and utilities as well as equipment vendors seem to bend the term such that it fits best their current product and service portfolios.

Besides marketing, the term *Smart Grid* is often associated with the increase of digitalisation and communication of power infrastructures. Integration of renewables, increase of efficiency as well as security of supply are major drivers. The discussion of the smart grid is not limited to technical issues like intelligent protection systems or smart metering.

Moreover, environmental as well as socio-political issues such as subsidiaries for de-central generation are subject of the smart grid. The smart grid transforms traditional power infrastructures into open platforms for providers, producers, consumers, service providers and prosumers.

As related work does not provide a universally valid definition, the following is an attempt to scope the smart grid. Instead of a precise and formal definition, it summarises key features and drivers:

Definition A *Smart Grid* is an electricity network which extensively uses Information and Communication Technology (ICT) to achieve an intelligent and energy efficient harmonisation of generation, storage, transmission, distribution, and consumption. It is an open platform for producers, service providers, and consumers of electrical energy and add-on services.

The increased deployment of ICT in power infrastructures bears many opportunities for traditional utilities to operate their infrastructures more efficient and stable. The Smart Grid, however, has also disruptive potential as it opens the infrastructure for new players and services.

3.3.1 Surrounding Conditions

A variety of surrounding conditions fuel the Smart Grid hype. Since conditions are often related to governmental motivation, e.g. unbundling and liberalisation, or based on immediate technical needs like increase in reliability in the United States, Smart Grid drivers vary greatly from region to region. In order to provide concrete and concise facts, the following focusses on European and in particular German energy markets. Information presented is gathered from "BMWi Energiestatistiken" (http://www.bmwi.de/BMWi/Navigation/Energie/energiestatistiken.html) and the European Technology Platform Smart Grid [41]. Drivers in other regions may be different. The conclusion, however, i.e. increase in ICT and deployment of intelligence in the grid, is true for all regions.

In Germany 50%, of plants today in operation were built between 1960 and 1980 and will meet their end-of-life in the next 5 to 15 years. Utilities are therefore pushed to find cost attractive alternatives for their generation infrastructures. In the year 1998, the "Energiewirtschaftsgesetz (EnWG)" introduced the liberalisation of energy markets. The regional monopolies of utilities is thereby abrogated, and transmission and distribution infrastructures must be opened for third parties. The maximum rate of return for energy

networks is capped such that third parties can utilise these infrastructures without disadvantages. The cap is determined by comparing expenses of different utilities. In the year 2008 the amendment added the liberalisation of metering services which entitles the customer to chose a metering provider different from the utility. Unbundling forces the utilities to separate networks and energy sales. Ownership unbundling goes one step further with the obligation that network and energy sales are even executed by two separate legal entities. Hence, unbundling and deregulation requires the traditional utility to increase efficiency in network operation and enhance IT infrastructures for seamless integration of cross-enterprise business processes.

The "Erneuerbare Energien Gesetz (EEG)" was implemented to advance deployment of heat and electric generators based on renewable sources. Its aim is to increase the percentage of renewable energies of the entire generation mix to 30% by 2020. By law, utilities are forced to buy electricity from renewable sources at governmentally fixed prices. Challenges for the utility arise once the numbers of de-central renewable sources increases since sources like wind and solar behave stochastically and cannot be controlled.

The volume in energy trading increases throughout Europe continuously, causing higher load on transmission infrastructures. Moreover, the expansion of offshore wind parks induces substantial load on the transmission network in Germany. The increased load causes congestion and reverse load flows which may circumvent protection systems causing destabilisation of the power infrastructures. Hence, already today, transmission infrastructures are heavily extended and further extensions are planned.

Per capita consumption of electrical energy rises continuously. Although household appliances like refrigerators, washing machines and dryers become more efficient, new devices, e.g. digital entertainment and electric vehicles emerge, which demand even more powerful infrastructures.

3.3.2 Challenges and Requirements for Smart Grid Deployment

In [41] the SmartGrid Advisory Council elaborates ten key issues that need to be addressed by Smart Grid in the short to mid term with regard to successful deployment. Associated with these issues are the key challenges and requirements for the Smart Grid. Similar issues are identified in related works. Below is a summary of challenges and requirements based, among others, on [70], [41], [71], [42].

1. Education of all stakeholders. Expansion and transformation of traditional grid infrastructures may be obstructed when individual stakeholders are not aware of the benefits the Smart Grid contributes to their utility. Hence, concepts, ideas and ben-

efits must be actively marketed. Additionally, it must be made clear that the new grid architecture is not a green-field solution but builds on existing infrastructures.

2. New planning and engineering for de-centralised grid architectures. New de-centralised architecture concepts are the best motor for the democratisation of power infrastructures. In order to deploy these new design, new engineering methods and tools are required.

3. Strengthening the grid integration to prevent disturbances. As the utilisation of the transmission increases throughout Europe, advanced integration is required. To maintain the stability of supply, wide area monitoring (WAM) and wide area control (WAC) solutions in combination with the ability to actively route load flows must be deployed at a broad scale.

4. Moving grids offshore. Large offshore generators like wind farms and wave or tidal-based generators require offshore networks to maximise the efficiency of generation and transmission.

5. Active users need active grids. Upgrading power consumers to prosumers requires active distribution networks for co-ordinated control and deployment. The major requirement is the availability of reliable communication infrastructures.

6. Adequate communication for new services and players. New market players like operators of virtual power plants, energy management service providers and meter service providers will emerge. A reliable communication infrastructure connecting all parties is required for data exchanges and technical support.

7. Enhanced intelligence for enhanced efficiency. Active demand participation will increase the efficiency of energy consumption iff a certain level of co-ordination of network, residential or industrial loads, users and manufacturers of home appliances is accomplished. Besides technical aspects, appropriate incentives must be offered in order for the new technology to be adopted.

8. For dispersed generation, dispersed storage is required. The intermittent and disperse characteristic of generators based on renewable sources requires efficient storage technology.

9. Mobility. Sustainable transportation like electric cars will have a major impact on the Smart Grid. Network design needs to allow for large mobile generation and storage. Additionally, a corresponding ICT infrastructure is required for seamless access to accounting and billing of energy services.

10. Initiate research on Smart Grid topics now. Extensive research is required to start immediately in order to deliver applications and solutions for the long term perspective of 2050.

This work, focussing on data-centric information and communication, particularly addresses items 5,6 and 9 as well as partially 2,3 and 7 of the above list.

3.4 Summary

Starting with essential concepts and components, this chapter introduced the power system domain which is chosen to evaluate a new architecture for data-centric communication. An introduction to power system control underlined the importance and necessity of communication for control and stable harmonisation of generation and consumption of electrical energy. The last section elaborated the Smart Grid. One the one hand, the Smart Grid involves the upgrade of old infrastructures with new digital equipment to achieve higher efficiency. On the other hand, it constitutes a new service platform to address the challenges power infrastructures will be faced with in the mid and long term.

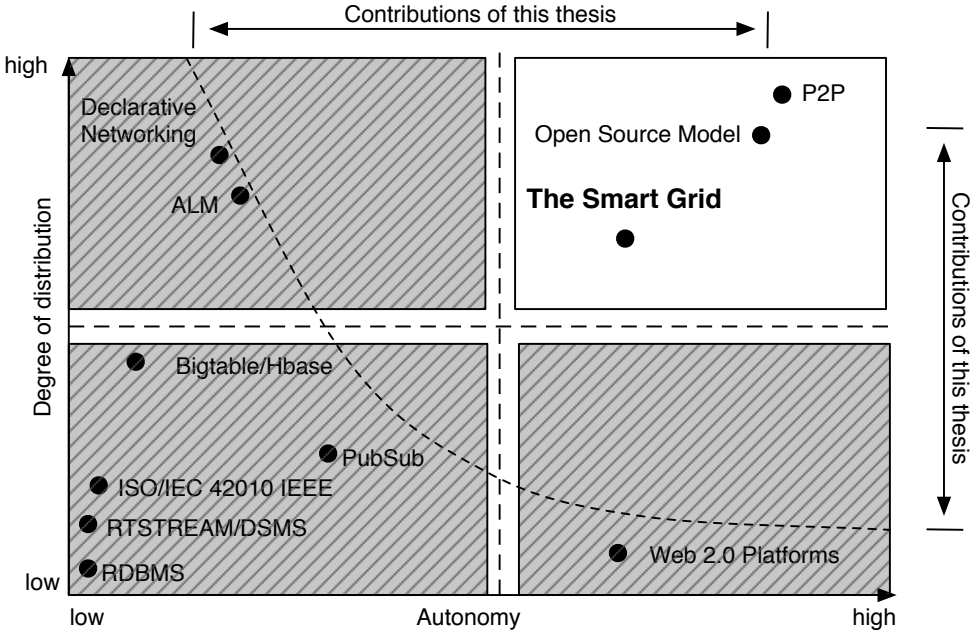


Figure 3.2: Technology cluster by degree of de-centralised organisation and autonomy

In the previous chapter (Chapter 2), technologies for large-scale data-centric systems have been introduced. The chapter concluded in Section 2.7 with a classification of technologies along two major dimensions namely degree of distribution and autonomy. By classifying

the Smart Grid along these two dimensions, Figure 3.2 complements this categorisation. This figure illustrates the suitability of technologies for the Smart Grid and exposes the technological gap between existing approaches and the requirements of the new systems. It also shows that the Smart Grid belongs to the class of large-scale distributed systems and that technologies identified in the previous chapter are promising candidates to address the challenges of the Smart Grid.

Before the major contributions targeting the technological gap are described in Chapters 5 and 6, the following chapter introduces the scientific methods and tools applied in this work. An architecture methodology is used to extract relevant requirements, define quality attributes and apply proven evaluation methods. Simulation strategies allow for large-scale examination and quantitative evaluation.

Chapter 4

Scientific Framework: Methods and Tools

This chapter introduces the methods and tools used to create and evaluate the scientific contributions of this work. In general one can distinguish between two approaches: *Empirical* approaches aim to infer a universal model by generalising from real-life observations. The validity of the model is verified by comparing its characteristics with empirical facts. In contrast, *constructive* or *rational* approaches develop a theoretical foundation to gain knowledge about the systems under investigation. While, in the empirical approach, real-life observations yield a system model, in the constructive approach they rather exemplify the correctness of the theoretical framework.

Applied to engineering sciences in general and complex distributed software systems in particular, both approaches raise difficulties. The dynamics of a complex technical system, such as a distributed software system, is determined, on the one hand, by the high variability inherent to components, e.g., heterogeneous hardware and different communication protocols, and, on the other hand, by the high level of connectedness of individual components. Following a constructive approach has the advantage that the correctness of the system can be analytically verified in context of the respective theoretic framework. To do so, the system as a whole or parts of it must be transferred in a form such that the analytical method can be applied. This *model*, however, must capture all aspects contributing to the dynamics of the real world system which is challenging if not impossible. Given that complex systems often behave chaotically, i.e. they are sensitive to minimal variations in the environment, the results generated in an abstracting model might not be relevant in the real world.

In an empiric approach, real world systems would be analysed and key requirements identified. Based on these requirements an architecture would be derived. While the solution,

if appropriate software engineering methodologies are applied, will yield a near optimal result for the concrete case, the disadvantage here is that the architecture is relevant to the one specific system only and often inflexible towards variations in requirements. Hence, the methods provided in this chapter are balanced between constructive and empirical approaches. This yields, on the one hand, real world applicability, while, one the other hand, general solutions that may be applied to a whole class of systems.

The chapter is structured as follows. First, the concepts of software architecture and modelling of technical systems are introduced. Based on this foundation, methods to create concrete architectures are presented. Section 4.3 elaborates on simulation methods and tools for large scale complex systems. Subsequently, a methodology to design domain specific languages is provided by Section 4.4.

4.1 Architecture and Model

The goal of this thesis is to develop an information and communication architecture for large industrial systems which enables them to operate robustly and efficiently over decades, while providing the flexibilty to react to changes in the operation environment and requirements. Unfortunately, the term *architecture* is used imprecisly in software engineering and hence needs clarification. Bass et. al. [11] provide a definition of a software architecture that is relevant for this thesis:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architectures provide both a *lingua franca* for all stakeholders and a technical blueprint for the system under investigation. To specifically address the foci of stakeholders, architectures consist of a variety of different structures also referred to as *views* which limit details to only essential information for an individual or class of specialists. Typical views are:

- the *functional view*, which establishes an abstraction of the systems' functions and their relationships.
- the *concurrency view*, which describes which processes and threads are created in the system and how they will communicate, synchronise and share resources.
- the *code view*, where the system is manifested as classes, objects, procedures, functions and their compositions in terms of a concrete programming environment.

- the *development view*, which provides a perspective on the structure of source code as a repository of files, directories or databases. It allows developers and maintainers to create, modify and manage code artefacts in a coordinated manner.
- the *physical view* or *deployment view*, describes the system in terms of hardware resources and the deployment of components to hardware.

The different views of the architecture developed in the thesis are manifested in several models of the system. Similar, to the term *architecture*, the term *model* can be found in a large number of definitions. It is derived from the latin *modulus* which translates to form, pattern or antetype. However, a transfer to the context of a software development process remains difficult. This can also be observed by looking at, e.g., the definition provided by the UML:

> Model: a semantically closed abstraction of a subject system.

The usage of the words *abstraction* as well as *subject system* are both not precise and it remains unclear what exactly is meant by *model* in the UML. A different approach is taken by Stachowiak [135] who describes a model by its inherent features which are:

- *Map feature*: Every model stands for something else, i.e., its original. How the original is related to the model depends on the interpreter. Stachowiak takes therefore a constructivist position which can be stated formally

$$O \triangleright_I M \tag{4.1}$$

  which reads: M is a model of O for observer I. The relation $\triangleright_I$ is $n : m$, i.e., an original can have multiple models and vice versa which is reflected by the different views of the architecture.
- *Reduction feature*: A model has only a subset of the original's features and model features may slightly vary in comparison to the original.
- *Pragmatic feature*: The purpose of a model is to replace the original, given certain conditions and goals.

As stated above, a model is a representative of its original and hence can be seen as a sign that points to the original. This perspective connects modelling with semiotics, i.e. the science of signs. In this context, models have, like signs, syntactic, semantic and pragmatic aspects. The aspects can be illustrated by the tetrahedron version of the semiotic triangle (Figure 4.1) as introducted by FRISCO Group [43].

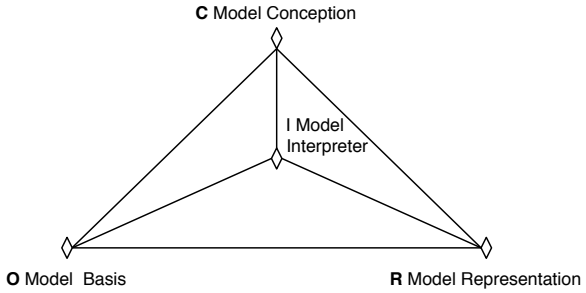


Figure 4.1: The Semiotic Triangle

The theoretical core of the model is routed in the model conception C which determines how the original is understood as well as which features are important and which can be omitted or must be added. Several representations can exist for one conception. However, for each representation only one conception exists. The pragmatic basis O for the model determines purpose, type and profoundness of modelling. A model becomes syntactic through its denotation R, which in this thesis, is accounted for by a description in the UML or other forms of diagram formalism as well as the programming language SCSQL, which provides a syntactic representation of component inter-dependencies and algorithms.

By establishing a common definition of the term architecture and its artefacts, i.e. models and views, this section introduced the principal concepts for methods to create and evaluate this work. The following sections illustrate how architecture artefacts are generated and thereby constitute the foundation of Chapter 7 which evaluates the contributions of this thesis.

4.2 Methods for Architecture Selection and Evaluation

A considerable body of research has been done on methods for software architecture decision and evaluation. A general consensus [11] is that the achievement of quality properties is key for a successful system. Therefore, methods like ATAM, SAAM or ARID [11] [26] proceed by identifying quality attributes, i.e. non-functional requirements, in so called quality scenario descriptions. A scenario captures a particular workflow or use case of the system. According to Bass [11], scenario descriptions are structured into six essential parts:

1. *stimulus*, i.e. a condition that needs to be considered when it arrives at the system
2. a *source of stimulus*, i.e. some entity that generates the stimulation

3. an *environment*, i.e. the stimulus is embedded into certain conditions
4. an *artefact*, i.e. the part of the system that is simulated
5. a *response*, i.e. the stimulus triggered activity
6. a *response measure* which defines how the effect of the response is measured

Having quality attributes identified, so called *tactics* and *architectural patterns* are used to create a design [123] [11] [18]. A tactic is a design decision to achieve one or more quality attributes. Conflicts may occur, e.g. a tactic may introduce redundancy to increase availability which, however, may influence performance and latency which might be another quality requirement. Generally tactics can refine other tactics, e.g. availability can be temporal, i.e. data available at a certain point of time and regional, i.e. data available at a certain place. Hence, tactics are organised in hierarchies. Moreover, tactics can be packaged into patterns, e.g. a pattern supporting availability might use both redundancy and synchronisation.

Quality attributes, tactics and patterns constitute the foundation for architecture evaluation. Recapitulating the business drivers for Smart Grids in Chapter 3 the design decisions are evaluated by validating it against the scenarios described in Section 6.1. Thereby risks, sensitivity points, and trade-offs are identified and their contribution to the overarching business goal elaborated. Where appropriate, alternative designs are discussed and the advantage of the design chosen is clarified.

4.3 Simulation

Technically, large-scale complex systems are difficult to analyse. On the one hand, this is due to the sheer size of a system which requires a multitude of costly resources as well as an infrastructure for large amounts of data to be collected and processed. On the other hand it is due to the fact that the environment cannot be fully controlled which hence may cause uncontrollable variations in results hindering reproducible scientific reasoning. The latter can be addressed by simulating the system in an artificial environment that can be fully controlled. Simulations are used extensively in this work to validate the behaviour of the architecture and algorithms developed. This section provides a definition of a system simulation and introduces key concepts. Banks et. al. [9] define simulation as:

> "A simulation is the imitation of the operation of a real-world process or system over time. [...] The behavior of a system as it evolves over time is studied

> by developing a simulation model. This model usually takes the form of a set of assumptions concerning the operation of a system. These assumptions are expressed in mathematical, logical or symbolic relationships between entities, or objects of interest, of the system."

A prerequisite of simulation is the creation of a model that describes the system. The underlying concepts used to describe the system under investigation are *entities*, *attributes*, *activities*, *events* and *state*. A key concept of the simulation is the *entity* which represents objects of interest. Each entity has *attributes* that determine their structure and behaviour. An *activity* is a defined period of time with a known length. The *state* of a system consists of the minimal set of variables required to describe the system at any point of time. Finally, *events* may cause state variables to change. The simulation model used in this work is based on *Discrete Event Simulation*. In this environment, system state variables change only at those points in time where events occur.

A unique challenge of this work is the simulation of systems with large amounts of entities. These simulations require considerable memory and compute resources which usually cannot be supplied by a single computer. Although several standard discrete event simulators are available only a few support computation on more than one compute node. In context of this work, a survey has been conducted to select the most suited simulator product. Thereby the following requirements were evaluated:

- *Networking*. The simulator needs to be able to simulate different network protocols at different layers of abstraction, i.e. at the application layer, e.g. overlays or TCP/IP or the physical layer, i.e. ethernet or powerline.

- *Dynamics*. For reasons of resource efficiency the simulator must provide means to add and remove entities during simulation time.

- *Extensibility*. The simulation needs to provide an interface to implement simulation specific extension, e.g. synchronisation protocols or workload distribution mechanisms which might be simulation specific. The interface should be either in the form of an API or plug-in interface. Open source products provide the highest flexibility in this regard.

- *Statistics*. The simulator needs to provide means to collect and aggregate statistics on all parameters related to the simulation as well as the execution of the simulation. The latter is of utter importance during development of a simulation as it is used to optimise the simulation code to achieve better performance.

- *Scalability.* The simulator needs to be able to scale to large simulations hence it should be able to efficiently make use of additional resources provided for the simulation.

- *Distribution.* If a single computer is not sufficient, the simulator should be able to distribute workload to a number of compute nodes. Additionally, it needs to provide means to synchronise the computation and collect and merge simulation results.

- *Platform independence.* Software and simulations were developed on Microsoft Windows, Mac OS X and Linux systems. Small simulations can be run on desktop PCs or Macs while larger simulations should be computed on a Linux cluster. Hence, the simulator needs to support all three platforms.

- *Documentation.* Many of the open source simulators were developed by academic institutions without commercial interest. Although published for everyone to download and use, often minimal to none effort is spend on documentation and support. Since simulation may consist of many thousands of lines of code which are bound to a specific simulator, migration from one product to the other is not easily possible. Hence, a minimum of support and documentation must be provided.

Table 4.1: Simulator Survey

Product	Platform	Documentation	Networking	Dynamics	Extensibility	Statistics	Scalability	Distribution
NS2	Windows Mac, Linux	manuals tutorials mailing lists	multiple protocols	yes	plugin source	manual	1	no
3LS	Windows Mac, Linux	publication	overlay network	no	-	event based	225	no
GPS	Windows Mac, Linux	publication	app, overlay network	no	source	file sharing	1000	no
P2PSim	Windows Mac, Linux	web site	single layer	yes		minimal	N/A	no
PeerSim	Windows Mac, Linux	publication tutorial	components network	yes	source component	custom	1000000	no
PlanetSim	Windows Mac, Linux	publication tutorial	application overlay, network	yes	source	none	100000	no
SmurfPDMS	Windows Mac, Linux	publication	application overlay, network	yes		event based	N/A	yes
Overlay Weaver	Windows Mac, Linux	publication tutorial	application routing	no		message based	4000	yes[1]
P2PRealm	Windows Mac, Linux	publication tutorial	overlay IO	N/A		query based	k.A.	yes[2]

In the area of P2P systems several simulators are available that seemed like a good fit for the requirements above. Hence the following simulators where evaluated: NS2 [99], 3LS [143], GPS [157], P2PSim [1], PeerSim [72], PlatnetSim [4], SmurfPDMS [64], Overlay Weaver [130], P2PRealm [77]. Table 4.1 summarises the survey. In conclusion, none of the surveyed products fulfils all requirements, hence to be able to conduct simulations at the required scale, a distributed discrete event simulator has been developed.

The simulator can operate on a single or on multiple computers. For the general case, the simulation developer does not need to care whether the simulation will be executed locally or distributed. However, the simulator allows to modify workload distribution methods such that the specific requirements of a particular simulation can be addressed. Simulation execution undergoes four major phases:

Initialisation

During this phase configurations are loaded and the simulator infrastructure is set up. The phase includes the connection to remote computers, the creation of simulation entities and the parametrisation of the simulation.

Distribution

Following the initialisation phase, in the distribution phase, the initial workload, i.e. the instantiated entities and the initial events are distributed to all worker nodes of the simulation infrastructure. The standard method for this phase is a uniform distribution of entities and associated events to the available workers. Developers can overwrite this method for the specific requirements of their simulation, e.g. server entities may require more resources, hence it makes sense to deploy them on dedicated workers while multiple client entities can be deployed on a single worker.

Simulation

While the first two phases prepare the simulation, the third phase constitutes the actual execution. The simulation phase starts when the master computer, i.e. the machine where the simulation has been initiated sends a start command to all worker nodes. The workers start the execution by pulling the first event from the event list. The scope of the event execution may be local, i.e. affecting only the state of the local entities, or it might influence entities on remote workers. In the latter case, event execution is wrapped in a message and sent over the network for the respective worker to execute. Resolving the network

address of the target entity to execute the event depends on the distribution method. In the general case where the workload is distributed uniformly a hash based method is used to associate an entity identity with an IP address. For more advanced mappings the simulation consults the custom distribution component to locate the simulated entity in the infrastructure.

To maintain consistency in the parallel computation model, the simulator synchronises the computations of all workers. Several synchronisation methods are supported. The *conservative* method is based on the original works of Chandry, Misra and Bryant [23] [7] and [95]. It ensures that no worker receives an event e with a timestamp t_e where $t_e < LVT$ and LVT is the Local Virtual Time of the worker. Execution is halted if events may arrive that violate this condition. To prevent deadlocks workers exchange information on the events of the current timestep. Receiving this information from all workers, a local decision can be made whether a particular event can be executed or execution must be delayed until results from another workers arrive. For each timestep n^2 messages need to be exchanged which, depending on the characteristics of the workload and the number of workers, may not be efficient.

In the *lookahead* method, each worker can execute several timesteps up to a time limit t_l without synchronisation yet with the guarantee to ensure consistency. The method operates by adding a small delay to messages sent from one entity to another. The approach is valid for simulations of networked system as communication between entities has always a latency > 0. Each worker maintains a synchronisation table containing a row for each worker participating in the simulation. A row contains the ID of the worker, the delay and a value t_{lw} representing the time limit computed with information from worker w, i.e. the LVT at time the last message was sent plus the current delay to be added. The t_l for a worker is set as the minimum of all values in the table. At time t_0 this t_l is the same for all workers but during the execution the limit is constantly adjusted. Thereby two cases are distinguished: (i) a worker A receives a message from B. A updates its synchronisation table and computes t_l as the minimum of the sum of the current minimal delay plus the time the message was sent. By this procedure it is assured that no message is received with a timestamp smaller than a message previously received from the same worker. (ii) when worker A reaches t_l, it sends a *null* message to those workers that are associated with the minimum value in the synchronisation table. The message contains the actual LVT. Receiving workers update their synchronisation tables and respond with their current $LVT - 1$ which is used to update the local synchronisation table.

Result Collection

The collection phase follows when all executions on all workers are finished. Simulation results and log files are retrieved by the worker that initiated the simulation. Log file entries are time-stamped, hence consistent result sets of the simulation for further analysis can be created. The result collection phase ends with a clean up procedure where local log files and temporary files are deleted. Afterwards the simulation infrastructure is prepared for the next execution.

4.4 Methods for Language Selection and Evaluation

The development of programming languages in general and Domain Specific Language (DSL) in particular, is a complicated and time-consuming undertaking. Difficulties emerge, on the one hand, by the requirement of both domain and language development expertise and, on the other hand, by the trade-off between the design of comprehensive but costly versus minimalist implementation with few domain-specific alleviations. Initially, it might be less than obvious that development of a DSL is worthwhile as benefits might only emerge after a considerable amount of programming in a General Purpose Language (GPL) has already been done. While in the latter case a DSL might still be useful for reengineering or software evolution purposes [14], when designed carefully, DSLs are able to bring benefits early in the software engineering process. A rich body of material and patterns, e.g., [133] [94], is available for the DSL development process which generally consists of (i) decision making, (ii) design and, (iii) implementation. In the following each phase is described in further detail.

In the decision phase, key questions regarding the expected benefit of a DSL need to be answered. Mapping these questions to patterns supports the decision process. Based on [94], Table 4.2 lists decision patterns relevant for the decision process in the context of this thesis.

With indicators for a beneficial development of the DSL at hand, design patterns are used to create a design for the DSL. Options in the context of this thesis are listed in Table 4.3.

As soon as the design of the language is created implementation can be started. Similarly to other phases in the development process, patterns constitute the basis for an implementation strategy.

Taking the patterns in Table 4.4 into consideration, several trade-off decisions must be made. Interpreter and compiler approaches allow notations close to domain expert tax-

Table 4.2: Decision Patterns

Pattern	Description
Notation	Adds domain-specific notation beyond the capabilities of the GPL. E.g. special operators, function notations, access of remote data, quality attributes of data items
Data structure representation	Complex data structures need to be initialised causing tedious and error-prone code fragments. A DSL allows for easier definition and initialisation of data types.
Data structure traversal	Traversals over complicated, e.g. large hierarchical, data structures can be expressed more compactly with a DSL
System front-end	A DSL based front-end can be used for handling system configuration and adaption

Table 4.3: Design Patterns

Pattern	Description
Piggback	A structural pattern which uses the capabilities of an existing GPL as a hosting base for the DSL. The pattern can be used whenever the DSL shares common language elements with the host GPL. Typically, the DSL is pre-compiled into the form a the host GPL. If the DSL is implemented as an interpreter a similar strategy can be implemented if the base interpreter allows to call it from within the DSL.
Language Extension	A creational pattern which adds additional elements of an existing language. In contrast to the piggyback pattern which uses the host GPL as implementation vehicle, the language extension pattern is used when a base language is enriched with semantic and syntactic elements to form a DSL.
Language Invention	In this pattern the DSL is created from scratch without relationship to any existing language.

Table 4.4: Implementation Patterns

Pattern	Description
Interpreter	DSL statements are interpreted in a standard fetch-decode-operate cycle. Interpretation is beneficial for dynamic languages or if execution efficiency is not the primary target. Compared to compilation, interpretation allows for easier extensibility as well as more control over the execution environment, e.g. for security reasons.
Compiler	The DSL is compiled to base language constructs and library calls. Since the compiler output is in the form of the base language all base language optimisations are effective. Execution environment control is limited but possible, e.g. through regulation of allowed transformation rules.
Pre-Processor	DSL constructs are translated to the base language prior compilation. This approach limits the DSL's flexibility to introduce operators and other semantics.
Embedding	The DSL is embedded into the host GPL by defining abstract data types and operators. While programmers can use higher level constructs they are still bound to the syntax of the base language.
Extensible Compiler	In the pattern the GPL compiler is extended with the constructs, operators, domain-specific optimisation and code generation of the DSL. Extending an existing compiler which was not specifically designed to allow for extensions can be time consuming and costly.

onomies. Since they control the entire translation process, they can provide detailed feedback on errors and exceptions. Domain-specific optimisation and transformation enables tailored solutions which might be beneficial especially in complex distributed systems. Compared to embedded approaches there are several disadvantages as well. First, there is the high development effort since complex language processors need to be implemented. Second, language extension is difficult to achieve since most language processors are not designed with extension in mind, which also relates to the first point as it typically further complicates the processors design. Third, DSLs following the compiler/interpreter implementation patters are more likely to be designed from scratch. Hence, the chance of incoherent designs is more likely. Embedded approaches, on the other hand, introduce a series of advantages. The development effort is considerably lower than for compiler approaches. Embedded languages can take full advantage of concepts already present in the host language. In addition, tools such as Integrated Development Environments (IDE) and tool chains can be reused for embedded DSLs. However, embedded languages have deficiencies with regards to syntax extensions as the host language does not allow arbitrary extensions. Error reporting is rather vague as the host language is not aware of the DSL concepts and finally domain-specific optimisations are hard to achieve as the translation process and execution environment cannot be controlled. Based on [94], Figure 4.2 shows the flowchart utilised to find an appropriate implementation style for the DSL developed in this thesis.

4.5 Summary

The methods and tools used in this thesis have three dimensions. First, architecture decision methods and modelling are used to create a software architecture for the systems under investigation. Second, we employ discrete event simulations and theoretic frameworks for the design of algorithms, description of component interactions and ensuring the achievement of quality attributes. Third, a development method for domain-specific languages based, on decision, design, and implementation patterns. In the context of scientific theory, the methods and tools implemented are based on constructive as well as empiric approaches. Where appropriate, i.e. models that are simple enough, analytic methods are applied to verify correctness and behaviour of algorithms. Scenario-based software engineering methods yield models of the system under investigation. Architecture evaluation methods provide qualitative metrics for the benefits and effectiveness of the developed architecture. Where verification on the actual object is not possible, e.g. proving correctness of a remote protection system, realistic simulations based on sampled data and/or reference cases provide the foundation for scientific reasoning.

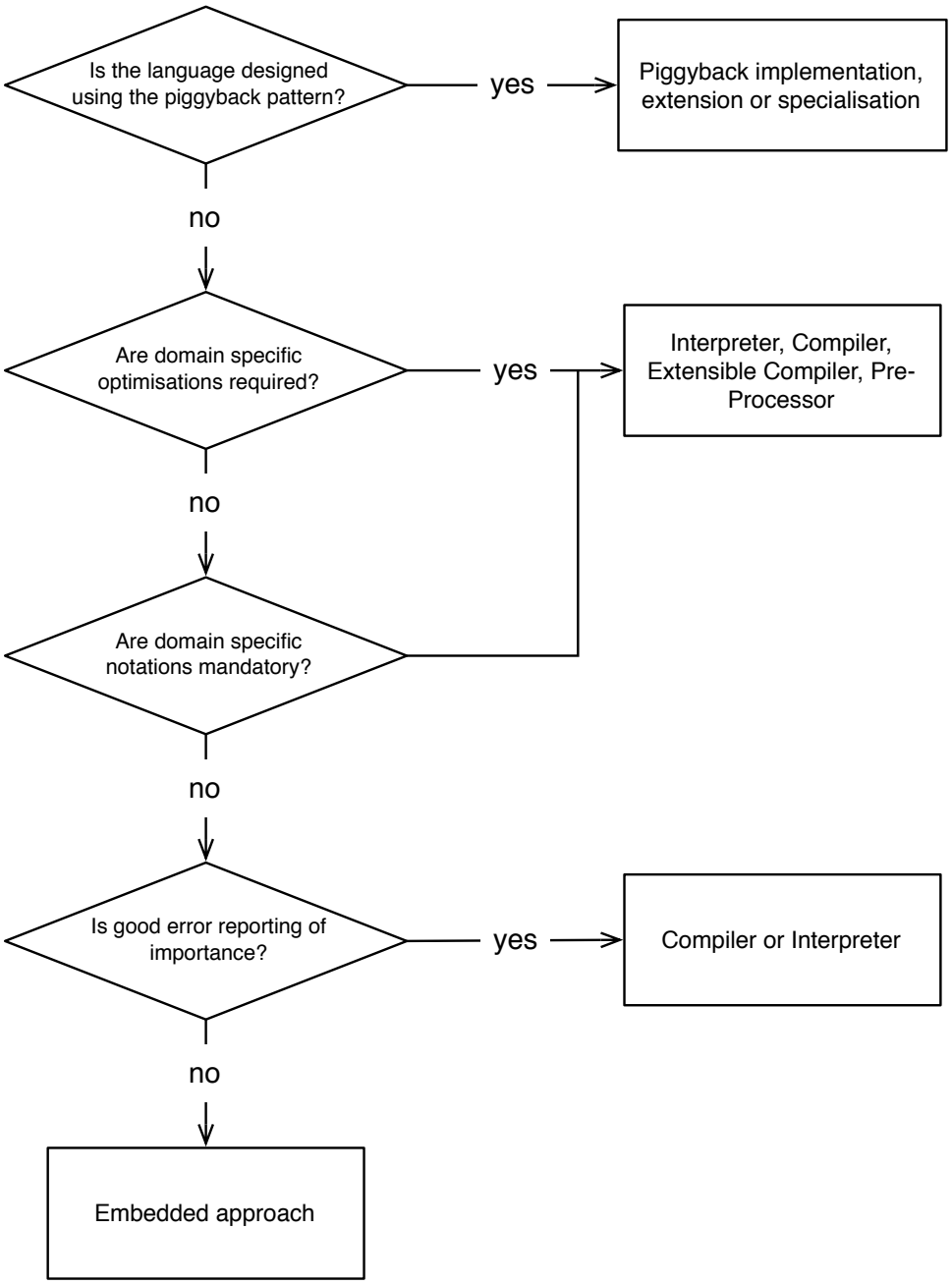


Figure 4.2: Implementation Pattern Decision

Chapter 5

An Ecosystem for Energy Services

Up to this point, we have elaborated challenges of future power systems, illustrated standard components of the power system infrastructure, and introduced the state of the art of technologies for large-scale systems. In this chapter the *Ecosystem for Energy Services* is introduced. It constitutes the underlying concept of this work and is the foundation of the architecture described in the following chapter (Chapter 6).

In [98], Northrop et. al. introduce the concept of the *ecosystem* perspective on technical systems by arguing that traditional, centralised engineering approaches are not adequate for highly complex and large-scale systems. In technical terms an ecosystem can be understood as a community of autonomous and competing components in a complex and changing environment. No matter whether they are natural or artificial, e.g. cities or the Internet, ecosystems are highly complex yet exhibit a high degree of organisation. Importantly, these features are not engineered but emerge naturally by local interaction during the evolution of the system over time.

As described in Chapter 1, new business drivers and Smart Grid technologies change the traditional utility business. In the process of transition, traditional power infrastructures grow and gain capabilities. New business models and players like the *Prosumer*, i.e. an end-consumer who sells home-generated electricity, emerge, causing system complexity to increase considerably. The ecosystem of energy services provides a metaphor aimed to cope with the problem of complexity, while, at the same time, allowing for reliable operation, adaption to future requirements, and exploitation of new business opportunities. The concept particularly addresses the research questions stated in Section 1.5.1: How can a system be designed that addresses all individual needs of its users and contributors? How can the system designed be evolved and adapted to changing policies and requirements?

The chapter proceeds by providing an overview of the metaphor followed by a description

of the basic components constituting the ecosystem. Subsequently, an abstract model for component interactions is introduced. In the following, data and network models describe the structure of the ecosystem. Section 5.6 introduces the policies and rules that determine capabilities and dynamics of entities in the ecosystem. The discussion focusses entirely on the concept level. The subsequent chapter (Chapter 6) will elaborate the technical implementation of the concepts.

5.1 The Ecosystem Metaphor

Biological ecosystems have evolved to high levels of complexity, sophistication, and sustainability through locally independent processes. In order to tackle the increasing complexity of large-scale systems, complexity sciences aim to utilise the ecosystem metaphor for technical systems as well [111] [98] [145]. This section introduces the ecosystem metaphor from a birds's-eye view. It describes entities of an ecosystem, the environment the ecosystem is situated in, and the principal driving forces that guide action within the ecosystem as well as adaptation to its surrounding environment. While this section concentrates on the abstract core concepts, subsequent sections further detail these concepts and establish the relationship with the technical aspects of smart grid power infrastructures.

The ecosystem for energy services is a community of people, organisations, tangible (devices) and non-tangible assets (data) in a complex and changing environment. Individuals and organisations follow their business goals and strategies, hence, act autonomously. In order to provide a concise description, the elements of the ecosystem, e.g. people or data, are henceforth referred to as *entities*. With this terminology the ecosystem can be understood as a *community of entities*. The commonality is determined by the set characteristics of an entity that is relevant in the ecosystem context. For instance, a measuring device and maintenance personal may share a commonality, e.g. location, in context of a maintenance process.

The ecosystem itself does not have an engineered structure but rather includes modifiable sets of rules and policies that regulate interaction and behaviour of entities in the system. It is an open system and therefore environmental forces[1] may have influence on entities and entity behaviour. The influences can be classified along three dimensions: first, an *economic environment*, i.e. determined by business drivers and competition among stakeholders, second, a *regulatory environment*, i.e determined by legal policies, and, third, a *technical environment*, i.e. dynamics of the power generation, transmission and distribution infrastructure (Figure 5.1). These surrounding conditions determine policy and rules

[1] Smart Grid Surrounding Conditions, Section 3.3

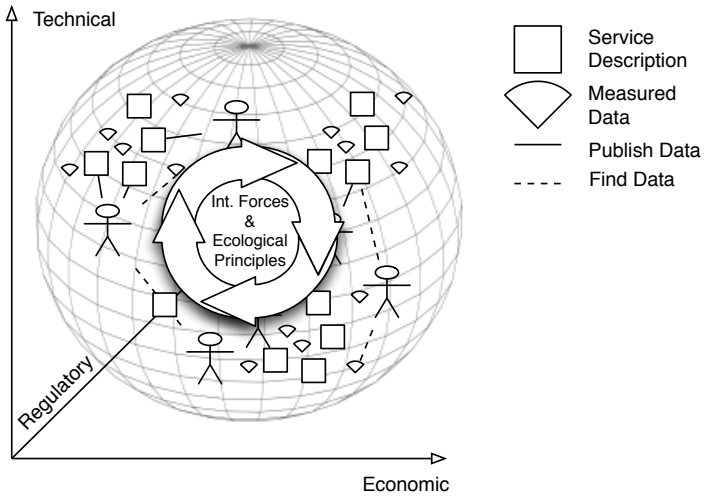


Figure 5.1: The ecosystem for energy services metaphor embedded in economic, regulatory and technical environments.

(Section 5.6) and correspondingly the technical requirements (Sections 1.3 and 6.1) for an architectural solution (Chapter 6).

Besides environmental forces, an ecosystem requires internal driving forces to regulate entity behaviour and interaction. Inspired by Adam Smith's 1776 publication "The Wealth of Nations" [132], the ecosystem for energy services uses market mechanisms to foster interaction and attractiveness. Smith identified three factors that constitute the free market: (i) the *pursuit of self-interest*, (ii) *division of labour* and (iii) *freedom of trade*. These fundamentals can also be identified in already established service ecosystems, e.g. in the public Internet. For instance, Google or Facebook: each log-in refers to a pursuit of individual interest. Without division of labour, the numbers of Facebook applications would be few and the Google index very small. Finally, freedom of trade is the very core essence that make e-commerce applications possible in the first place. The ecosystem of energy services implements the same three factors to foster communication and collaboration of entities within the ecosystem as well as owning and operating counterparts outside the system.

Applying the ecosystem metaphor to a technical system means that the system implements the ecological principles [145] [158] [76] of (i) *Evolution* to be able to adapt to new requirements and environmental changes, (ii) *Connectivity* to establish a community among participating components, (iii) *Commodity*, which reflects that the system key components are always conveniently available, (iv) *Flexibility* which expresses the ability to conduct alternative actions to meet a set goal and (v) *Diversity* which allows for the contribution of individual players with differing business goals and technical requirements.

The following Sections 5.2 - 5.6 provide the foundation for a technical implementation (Chapter 6) of these principles.

Value creation within the ecosystem is non-linear, i.e. in contrast to a linear value chain, value in the ecosystem develops by various interactions of networked entities. The value of the ecosystem increases for a particular entity A with the existence of one or multiple other entities B_i (see also Appendix C on value networks). Similar principles are effective in open service platforms of the Internet which provide services to collaborate, negotiate, conduct business and exchange information. In this context, the ecosystem provides a restricted set of core services (s. Section 5.2) to foster the driving forces explained above.

This section introduced the main characteristics of the ecosystem for energy services metaphor. The following sections will further detail the concept and provide the foundation for a technical mapping to the concrete building blocks and software artefacts that are developed in Chapter 6. A key element of this mapping is the *data* that is generated, exchanged, and used to describe entities and their services. Therefore data model, description, and modification are particularly emphasised. In technical terms this means that any entity is represented in the ecosystem by a *data item* (s. Section 5.4 and Chapter 6).

5.2 Core Services

In biological ecosystems the laws of physics define the rules of entity interaction and condition. Additional to the fundamental rules that were described in the previous section, ecosystem entities require a set of elementary capabilities to perceive and operate under the physical rules set. Similarly, in order to operate and participate in the ecosystem, a set of core services is required for the ecosystem for energy services. Core services are infrastructural components that deliver the functionality: (i) to differentiate between different entities and establish the connection to objects outside the ecosystem, (ii) to publish, search and discover entities and (iii) to provide incentives for service providers to offer their services in the ecosystem. Core service functionality is essential for the ecosystem. Without core services the ecosystem cannot operate. Although the functionality must be present, it may not be bound to a specific component or technical implementation, e.g. a web service. In the following each of the elementary services is briefly introduced.

5.2.1 Identification

The identification service provides means to uniquely identify entities within the ecosystem. Identification is mandatory to apply security mechanisms, provide traceability and verification, as well as engage in communication and general interactions. The identification service establishes the connection between objects situated outside the ecosystem and entities within the ecosystem. To concretise the concept of the identification service, consider the log-in at an e-commerce site which uniquely identifies the customer inside the shop, i.e. the ecosystem.

5.2.2 Registration

In order to advertise entities within the ecosystem, entities use the registration service to publish information describing their features and capabilities (s. also Figure 5.1). Providing means to search entities, a registration service achieves the mapping between a (non-)functional description of an entity and its unique identification.

Following the concrete example provided by the description of the identification service, the registration service is equivalent to a product catalogue which can be browsed or searched, given keywords or other criteria.

5.2.3 Incentive

This service provides incentives for service offerings. The ecosystem attractiveness increases with the number and diversity of its entities. By providing an incentive, more participants are attracted to participate in the ecosystem which, in turn, gains on attractiveness itself. By allocating a certain share of paid incentives to the incentive service provider, a business model is established where all players, i.e. incentive service providers, service providers and service consumers can participate beneficially. Similarly, within the ecosystem, incentives provide means for co-operation and support. Incentives can be monetary, units of resources or non-functional like priorities, or access to functionality or special nodes (s. Section 6.3 on the communication module).

In the context of the e-commerce example, the incentive service constitutes the payment service such as provided by credit card companies and Internet based payment services. The incentive mechanism builds upon the identification and registration services. It is of fundamental importance for a successful implementation of the ecosystem.

In context of the ecosystem metaphor, the core services provide the foundation for the

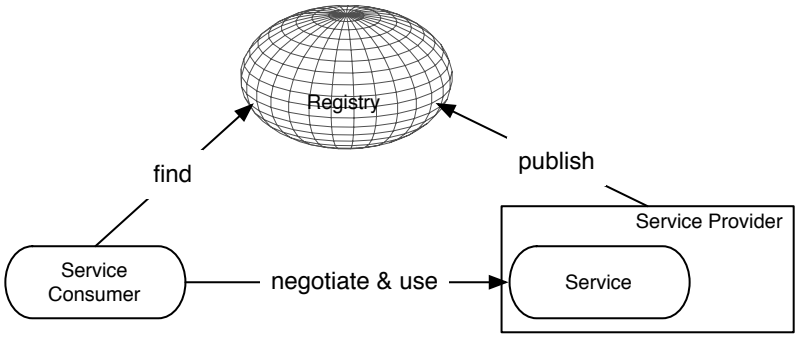


Figure 5.2: Service model

market mechanisms, i.e. the internal driving forces: (i) the *pursuit of self-interest*, (ii) *division of labour* and (iii) *freedom of trade*.

5.3 Interaction and Actor Model

In order to achieve their goals, entities need to interact with each other. This section provides the general patterns of interaction between entities. Underlying the interaction model is a simple actor model consisting of *consumer*, *provider* and *registry*. A consumer is an entity potentially interested in a service (functional) at a certain quality (non-functional). A provider publishes his capabilities and functions such that consumers can initiate a business relationship. The role of the registry can be assigned to any entity participating in the system. However, different registries with varying scope, e.g. local or global, are also possible (s. Section 5.5).

Ecosystem entities describe offerings, in case of people and organisations, and functions, in case of devices, by sets of functional and non-functional attributes. Entities publish their service descriptions in the ecosystem for service consumers to be found. In order to use a service, a service consumer states his interest in a service query and issues the query in the ecosystem. If a description matches the attributes specified in the query, the service consumer contacts the service provider and starts negotiation of the service relationship (Figure 5.2). Besides quality and function properties, the service description includes contact information of the service provider. However, specific details regarding negotiation are not part of the description but are rather exchanged during initial contact.

Service queries can be either entity-centric or content-centric. An example of an entity-centric query is: *"request the service provided by entity E"*, whereas a content-centric query could be: *"request a service where attribute A lies in a circle with radius R around a centre C"*. This interaction model is non-hierarchical, hence it allows for direct, peer-

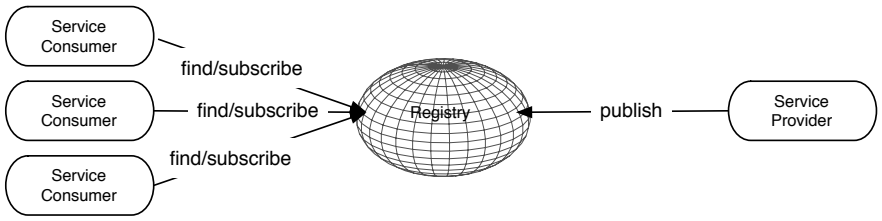


Figure 5.3: Data-centric interaction

to-peer interaction. This reflects the nature of most Business to Business (B2B) as well as Business to Consumer (B2C) scenarios.

In a simplified version of the model, e.g. when the service description is equivalent to the result of a service call (discovery query), the negotiate and use step can be omitted. In this case service consumer and provider are completely de-coupled and never interact directly (s. Figure 5.3). This is especially beneficial when multiple consumers request the same service because the provider does not need to negotiate with each of them.

A query can be *snapshot*, i.e. requesting a single description, or *continuous*, i.e. repeatedly requesting a description over a period of time. A continuous query on a description that delivers results only when the description changed is also called a *subscription*. By using subscriptions, entities can respond to changes in the environment like better or more suitable services, new measurements, or failure signals.

The interaction and actor model supports the ecological principle of *connectivity* and *flexibility*. It provides means to model entity relationships in a loosely coupled manner.

5.4 Data Model

The concept of data is integral in the ecosystem. Being the lowest common denominator, the data model must be minimalist yet expressive at the same time. In enterprise application integration the term "canonical data model" is used to describe a *design pattern* used to communicate between different applications. Instead of translating data formats between applications point to point, each application, if required to communicate with other applications, transforms its data to a canonical format understood by all applications. In this context the data model introduced in this section can be understood as a *"lingua franca"* or "canonical data model" for all energy services associated with the ecosystem.

The approach facilitates the flexibility of services to have a data model that fits best their requirements. Yet exchange between all services can be established by writing a single translator towards the canonical model. Hence the canonical model supports the loose coupling between services and makes a clear separation of responsibilities and domains possible.

Due to the openness of the ecosystem design, a particular challenge for this work is the design of a canonical data model while having limited knowledge of the services and their requirements. Standards like the IEC 61850 [68] provide rich object and service models for the Transmission & and Distribution domain. Naturally they are not suited for other domains such as business processes and cross-enterprise collaboration. On the other hand, integration standards like the OAGIS [100] are not suited for high performance power centric applications. Hence, existing standards cannot be used as blueprint for the ecosystem data model.

The approach chosen, therefore, is oriented towards file systems and object stores yet clearly abstracts from bits, bytes and the block level. Additional semantics can be implemented on top of the data model. This approach has two advantages, first, the openness for a great variety of services to participate in the ecosystem is taken care of and, second, services and applications can implement their own data model and add additional domain specific semantics on top of the canonical model. Reducing the canonical model to the absolutely required minimum, makes it applicable to all services and at the same time avoids the complexity of an all-embracing standard.

The model defines simple structures, entities and elementary types. It is transparent to any specific physical infrastructure, physical location and supporting hardware. The entire data contained in the ecosystem is comprised of a set of *data items*. Each data item consists of three parts, namely an unique *identifier*, a set of *metadata* and the actual *content*.

- The identifier is a binary array of fixed length. Each data item has exactly one, unique identifier. The identifier is immutable and cannot be swapped between two data items.
- The meta-data describes type (s. Section 5.4.1), quality (s. Section 5.4.2) and structure (s. Figure. 5.4) of the data item content. Meta-data may change during the life-cycle of the data item, e.g. in case of a sensor reading the quality may degrade with time.
- The data item content is the raw data as provided by the entity. In addition to the information provided by the identifier and meta-data, consumers of the data item need to know how to interpret the content.

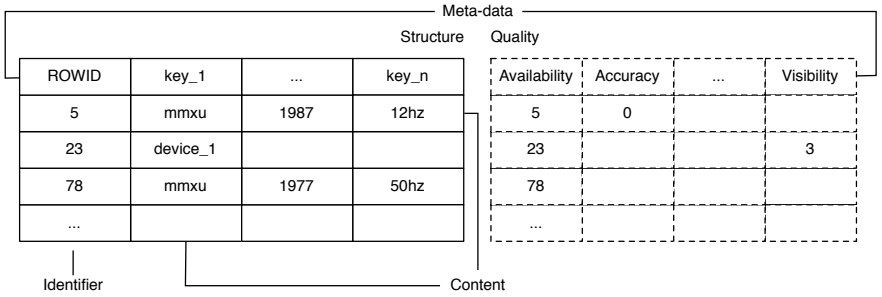


Figure 5.4: Table model

Data items are organised in *tables* (Figure 5.4). Similarly to the concept of Google's Bigtable [24], tables in the ecosystem are sparse with a potentially unlimited number of columns. A table description comprises a set of columns as well as quality attributes which describe the non-functional attributes of its rows. Data items are rows of tables. The data item identifier is a special column named *ROWID*. While a Bigtable is intended for large volumes of data, data items and tables in the context of this work are assumed to be rather transient, e.g. measurement data, with a short validity window or low volume, e.g. service descriptions.

Besides structuring data, tables also provide qualitative, e.g. temporal, views on data items. As an example, consider a table specified to contain measurements no older than ten minutes. This constraint is enforced by continuously checking rows, purging those outside the validity window.

Tables can have local or global visibility (see also Section 5.5). They can have an arbitrary number of rows. There are no restrictions on how many tables can be created other than the resource capacities of hosting environment.

5.4.1 Data Types

The ecosystem supports three atomic types, namely two numeric (*Integer*, *Double*) and one *String* type. Additionally, applications can define custom complex data types using the data item content. The Integer type is signed and exact while the floating point type, Double, is signed but approximate. Integer values range from -2^{31} to 2^{31}, Doubles are 64-bit double precision according with IEEE 754 [69]. Strings are of arbitrary length and can include unicode characters. Listing 5.1 provides an example of basic type declarations.

Listing 5.1: Type Delcarations

```
//Integer
INT i = 23

//Double
DOUBLE d = 0.4711

//String
STRING s = "Hello, World!"
```

5.4.2 Quality Attributes

As part of the meta-data, quality attributes make data items first class citizens in the ecosystem and hence are a key concept of this work. They determine the lifetime, discoverability, accuracy, and associated security of an item. The following paragraphs detail the different quality classes.

Lifetime

Data items can be either transient or persistent. The lifetime of a transient item can be controlled by quality attributes. The *validity* of an item defines a time window where the item is valid. Outside the window, the item has no purpose and will most likely be deleted. The lifetime of an item is indirectly influenced by external, unforeseen events such as failures and exceptions. The *reliability* attribute provides a measure of the effort to prevent the loss of an item. This may happen by replication, storage on especially reliable hosts or other techniques.

Discoverability

A key feature of the ecosystem is the search mechanism to locate any data item regardless of its physical location and the time the search was initiated. Additionally to the reliability attribute, the *availability* attribute determines the effort to be invested to maintain references to the item. This may cause the item to be indexed at multiple indices, additional meta-data to be indexed or the item to be cached at various locations. On the other hand, the *visibility* of an item declares whether the item is discoverable locally where it is stored or globally throughout the entire ecosystem. Depending on the scope of visibility, different techniques are chosen to publish the item. Hence, the visibility attribute affects the efforts required to achieve reliability and availability.

Accuracy

Data items can represent facts at different levels of precision. For example, consider a sensor capable of delivering readings at high frequency. Depending on the underlying network and the type of data, this stream of readings might cause high load on the infrastructure. Applications not relying on the provided level of precision can average several readings into one item and adjust the *accuracy* appropriately. Similarly, in an overload situation, applications aiming to compute an aggregate might skip certain readings to maintain responsiveness and provide indicators with lower precision.

Security

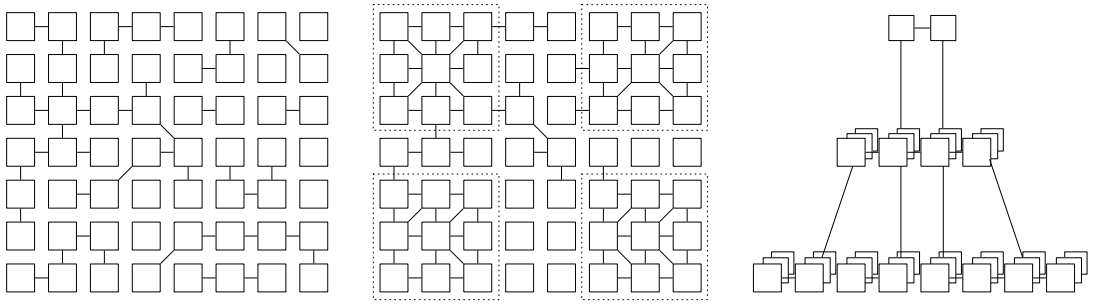
The content of a data item might include sensitive or critical information. The *confidentiality* attribute determines how and by whom the item can be changed or accessed. Specifying the *integrity* attribute allows for detection of a manipulation attempt. Finally the *non-repudiation* attribute causes every access, i.e. type of access and actor, on the item to be recorded such that the history of changes can be investigated.

The data model supports the ecological principles of *evolution*, *commodity* and *connectivity*. The table-based model allows to structure data yet by the possibility to add arbitrary number of columns, data schemata can be evolved without breaking existing semantics. The data model is designed as lingua franca and as such provides a consistent model for exchange between entities.

5.5 Network Model

The ecosystem is a community of entities (s. Section 5.1). The community structure is determined by interconnections between entities. In this section a network model is presented which provides the basic concepts that are the foundation of the community of entities and its structures, respectively. Similar to other sections, the discussion is entirely on a conceptual level and independent of physical networks, e.g. IP or Ethernet, or other physical infrastructures.

In the following, the set of entities and their connections is referred to as the *network* while entities are called *nodes* henceforth. A network consists of a set of nodes $N = n_0...n_i$ interconnected by a set of links $L = l_{i \times j}$ with $i \neq j \in N$. Nodes are either directly or transitively linked, i.e. if A is linked to B and B to C, then A is also linked to C via B. A network is fully meshed if every node has links to all other nodes. A set of nodes belongs



(a) Unstructured (b) Groups (c) Hierarchy

Figure 5.5: Network structures

to the same network if either direct or transitive links between any two nodes of the set exist.

A *group* or *cluster* of nodes (Figure 5.5b) is a subset of nodes which are highly meshed relative to the degree of connectedness inherent to the entire network. Groups allow to cluster the network into different areas of interest. Network partitions, in turn, can be inter-linked by so called *group links* thereby forming a network of groups. This partitioning process can be carried forward recursively. By associating each iteration with a *layer* in a multi-layer structure, hierarchical structures can be created (Figure 5.5c). As an example, consider the clustering of nodes according to their geographic location. At the lowest level, nodes having a certain distance to a given location, e.g. a city centre, belong to one group. At the next level, the distance is increased, e.g to cover a certain region. At the highest level the area covers all nodes. By traversing the hierarchy national, regional or urban areas can be addressed.

The network model implements the ecological principle of *connectivity*. It allows individual entities to form a community and structure their relationships.

5.6 Rules and Policy

In previous sections, the core concepts of the ecosystem have been introduced. After providing the overall picture in Section 5.1, system entities and their models of interaction were introduced. Subsequently, the data model section presented the lowest common denominator for entity interaction and exchange. The last section provided the network model which defined the concepts necessary to describe the organisation of entities. In this section the framework is introduced that stitches the individual elements together to a whole.

Depending on capabilities and properties, entities have a discrete set of states describing their possible conditions. At each point in time t an entity is in exactly one state S_t. Entity behaviour can be described by a *policy*, i.e. a set of rules, $P = r_0..r_i$, where each rule processes the current state of the entity and transfers it to a new state, i.e. $r_i(S_t) \rightarrow S_{t+1}$. The validity of a rule can be limited to the entity it is executed on. Henceforth such rules are referred to as *internal rules*. Rules may also inflict remote state changes on other entities. Rules that have influence on all entities are called *global rules* and rules that act only on directly connected entities are called *local rules*.

5.6.1 Rule Specification

Rules consist of functional and non-functional specifications. The functional specification defines (i) the subset of information that is processed, (ii) the processing goal and (iii) the state change of the executing entity once the rule is processed.

Listing 5.2: Rule Prototype

```
S -> SELECT F(X)
       FROM INFORMATION
      WHERE FUNCTIONAL CONSTRAINTS
    QUALITY ATTRIBUTES
  RECEIVERS <RECEIVERSET>
```

Listing 5.2 depicts the skeleton of a rule. Similarly to SQL, “SELECT ... FROM INFORMATION” selects columns and tables to be processed by the rule. The “WHERE FUNCTIONAL CONSTRAINTS” clause restricts the information using equality predicates and arithmetic expressions. The set of “QUALITY ATTRIBUTES” constrain the information subset to meet certain quality goals (s. Section 5.6.2). Finally, the processing goal specified by “F(X)” transforms the current state to a new state by setting a view “S->” on the processed information. Although the rule is evaluated at an individual node, the state change may not be restricted locally. The assignment “S@node1”, for example, would set the view on a node specified by “node1”. Similarly, the set of “RECEIVERS” specifies a set of nodes that receive the rule result which is labelled by the assignment. An concrete example is provided in Listing 5.3.

Rules can be executed either once or repeatedly. Repeated execution can be specified by either providing a number of iterations or by providing a sequence of points in time. While the first option belongs to the functional description, the latter is able to determine the quality of the execution.

Listing 5.3: This example rule modifies S to the sum of X of table measurements where X has a value smaller than 10 and corresponding items are valid for at least 20 seconds. The result set is also send to a node identified by NODE12.

```
S –> SELECT SUM(X)
       FROM MEASUREMENTS
      WHERE X < 10
        AND VALIDITY > 20s
  RECEIVERS NODE12
```

5.6.2 Quality Attributes

Similarly to data items, rule specifications include non-functional constraints. These constraints influence not only the targeted information subset, but also how the rule is executed. Rule quality attributes can be classified into three classes namely: *performance*, *reliability* and *security*.

Performance

Performance related attributes include *latency*, specified as the time between rule execution is initiated and the actual execution starts, *throughput*, in case of continuous rules, specified as the number of executions per given time interval and *response time*, specified by a time period from initiation to the state change.

Reliability

Reliability is the ability of the (rule) execution to perform certain conditions for a specified period of time. It is specified as either *high*, *low* or *don't care*. Thereby each value refers to the mean time between failures (MTBF) for the executing entity. It may cause the increase or decrease of execution priority for the rule. In case of local or global rules it may trigger the migration of execution from one entity to another, more reliable, entity.

Security

For local and global rules the security attribute, *trust*, determines the location of execution. Trust is specified by a *trust level*, e.g. *internal*, *external*. Trust levels can be defined according to needs of the intended use. If the trust level cannot be matched rule execution fails.

The policy and rule model supports the ecological principles of *evolution* and *flexibility*. State changes can be specified declaratively and hence are not bound the specific details of technical implementations. Rules can be modified to reflect changing requirements and environmental conditions.

5.7 Summary

This chapter described the conceptual framework for this thesis named ecosystem for energy services. In Section 5.1 the ecosystem metaphor is explained and the key principals (evolution, connectivity, commodity, flexibility, diversity) and driving forces (self-interest, division of labour, freedom of trade) for entity behaviour and interaction were introduced from a bird's-eye view. Subsequent sections detail these core concepts and describe the implementation of the key principals in several models. The models constitute the foundation of the technical implementation in the following chapter.

The concept of evolution is implemented by the flexible data model (Section 5.4) and modifiable rule and policy model (Section 5.6) which allow for constant adaption to new requirements without breaking existing relationships. Connectivity is manifested by interaction (Section 5.3) as well as the network model (Section 5.5) which allows various structures of relationships. Commodity results from the design of the data model as *lingua franca* between entities. Flexibility is implemented by the interaction model allows for a loose (re-)coupling of entities and the rule and policy model which allows declarative specification of state transitions. Finally diversity is supported by the data model which does not restrict entities to a unified and global model but rather allows to maintain a domain specific local data model and translate those parts relevant for exchange with other entities towards the canonical model.

To incorporate the driving forces: *self-interest*, *division of labour*, and *freedom of trade*, the ecosystem provides three core services: (i) identification, (ii) registration and (iii) incentive. The following chapter provides a concrete implementation of the concepts in form of a data-centric information and communication architecture.

Chapter 6

A Data Centric Architecture for Large-Scale Industrial Systems

In Chapter 1, challenges and general requirements of a new architecture for power systems have been discussed. The previous chapter (Chapter 5) introduced the concept of a data centric ecosystem, which aims to integrate distributed entities and ensures the availability of information at the right time and place. This chapter concretises the concept by describing a software architecture for the ecosystem using the methods outlined in Chapter 4. The architecture is embedded in the Smart Grid environment as elaborated in Chapter 3 and extends the current state of the art as presented in Chapter 2.

This chapter is organised as follows: two scenarios are depicted as means to determine architectural tactics and requirements. Subsequently, the overall architecture and its key components are provided, before the next sections detail individual components and their interactions. A dedicated section illustrates the capabilities of the query language and its compiler architecture. Selecting key components, the last section establishes a implementational view on the architecture and gives insight on a prototypical implementation.

6.1 Scenarios

To create an architecture for large scale industrial systems, in this section two scenarios, namely *backup protection* and *vertical information integration* are investigated in further detail. Scenarios are chosen carefully to be representative for the key components of the architecture which are (i) the *indexing cloud* to ensure global discoverability and integrability of data and (ii) the *networked query processors* for in-network processing enabling real-time availability of data as well as accomplishing the task of transformation

from raw data to information. Each description starts by providing the general context and use case first, followed by a selection of concrete sub-scenarios. Using the approach outlined in Chapter 4, quality attributes are extracted and appropriate tactics for their achievement are identified.

The general use cases are rather complex with a multitude of actors and responsibilities. The descriptions provided are at a fairly coarse gained level and hence are neither comprehensive nor complete but prioritised on key aspects and challenges that are in focus to this work.

6.1.1 Scenario 1: Remote Backup Protection

Protective relaying systems are usually built with various levels of redundancy to ensure isolation of fault conditions and equipment quickly. While local backup systems, i.e. redundant relay devices, are easily disabled by severe component failures, remote backup systems provide additional robustness by physical separation. Remote backup systems are configured to sense abnormal states, e.g. voltage peaks, on transmission lines. In case of a fault, the remote backup system waits for a configured amount of time before it becomes active. During this time period, the primary protection system has the chance to clear the fault. Hence, if the fault is not cleared within this time period, backup protection devices can assume that the primary system is malfunctioning. The backup protection system will then operate circuit-breakers to isolate transformers and other substation equipment.

Remote backup systems often lack important information to fully assess current network conditions. Thus, they often react incorrectly and either trip a healthy transmission line or fail entirely to isolate a faulty line. This behaviour may yield catastrophic cascading events and even blackouts. Moreover, unnecessary mechanical stress on switch gear has considerable effect on the lifetime of equipment and hence yields increased cost. By providing additional information from all protection relays participating in a protection system, control nodes can analyse the situation and issue correct actions quickly in order to prevent catastrophic failures. The scenario covers two aspects: first, engineering of a standard backup protection system and, second, supporting advanced functionality by providing additional state information to the control devices of the backup protection system.

Description

Figure 6.1 displays the example system which is considered for the scenario description. The system is composed of six substations, A-F, and seven circuit breakers 1-7. A flow

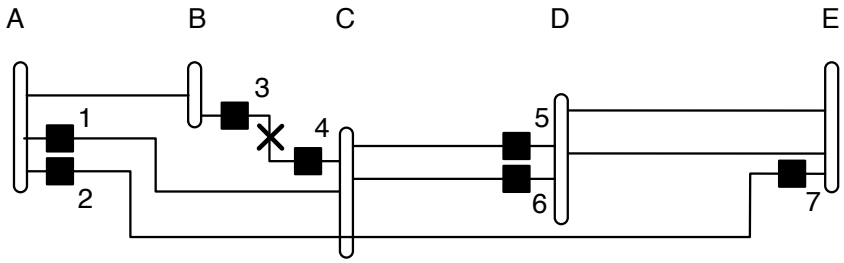


Figure 6.1: Example system with substations A-F and circuit breakers 1-7. Simulated three phase fault between B and C. Based on [74]

of power from left to right is assumed. In order to provide a concise description, in the sequel, it is focussed on key entities and flow of information. Power system details are omitted.

A three-phase fault, e.g. a short circuit of a transmission line, between substation B and C is investigated. In the normal case, the fault would be detected and correctly located by the primary protection system which would cause breakers 3 and 4 to isolate the line in about 50 ms. In case of a failure, e.g. when substation C fails to disconnect the line, breakers 1,2,5,6,7 would open which would disrupt the entire transmission corridor. Incorporating additional information on power flows and trip recordings in the control decision, allows to block or delay breakers 1,2,4,6,7 until the situation is fully assessed. Having detected the hidden failure, related equipment, i.e. breakers and their control nodes, are then able to issue an open command to breaker 4 thus isolating the faulty line while leaving the healthy lines energised.

Engineering such a system is complex and costly as all involved equipment is connected point to point, i.e. relays in substation A have statically configured communication links to relays in substation B. Hence replacement of individual equipment often requires configuration changes in several other devices. Since devices may be from different vendors, different configuration tools have to be used and configuration files have to be converted for importing it into other tools. Even with emerging standards such as IEC 61850 [68] which include standardised engineering processes, slight deviations in the interpretation of the standard or only partial implementations are still the norm rather than the exception, hence causing these problems to persist. Moreover, due to very long life cycles of energy automation equipment, heterogeneous installations are common which include device generations separated by decades. Generally, the engineering process is executed manually, which is costly and leaves enormous potential for mistakes. The case is further complicated if substations and breakers are operated by different legal entities.

As investigated in [74], a controlling node has about 200 ms for analysing data, assessing the situation and issuing respective control commands. Therefore, efficient communication paths must be configured and control nodes must reserve enough resources to process a request timely.

Scenarios

The description above contains three concrete scenarios which are onwards referred to as scenarios 1a-c. *Scenario 1a*: the protection function, i.e. the processes of sensing a fault, disseminating the information, evaluation, and control decision. *Scenario 1b*: the process of engineering and maintaining the protection system. This includes discovering participating devices, detection of communication links and deployment of protection algorithms. *Scenario 1c*: the access control for (i) the protection information, i.e. measurement and operation command and (ii) the deployment and modification of control code.

Quality Attributes

Table 6.1 summarises the scenarios elaborated in the previous paragraph using the notation described in Chapter 4. From the description, the following quality attributes are derived: Performance, predictability, modifiability, adaptability and security.

- *Performance.* The time required to generate a response is highly critical for successful operation of the system. If, based on the sensed situation, a decision is delayed, equipment damage or even harm to human life may be the consequence.
- *Predictability.* If a device participates in the remote backup protection system, it must behave deterministically. A unit must be able to guarantee that a response is generated within a specified time window. Not meeting this constraint renders the unit contribution useless as the response cannot be processed and will be discarded.
- *Modifiability.* Automation devices are deployed to operate for decades. The environment and the requirements of the protection system, accordingly, may change over time requiring adaption to the new situation. Modification includes deployment of new protection algorithms as well as rules to discover and connect to new devices that participate in the system.
- *Security.* Allowing modification of the protection functionality facilitates malicious manipulation of the devices. Furthermore, opening networks for third parties, e.g. for other Transmission System Operators (TSO) on tie lines, makes devices applicable for denial of service attacks which may interrupt the protection functionality.

- *Adaptability.* Although protection devices can be assumed to be very reliable, increasing the number of entities in the system also increases the probability of failure. Individual devices need to adapt to internal as well as external conditions and their dynamic change over time. The quality attribute of *Adaptability* can be further classified as
  - *Scalability.* The ability to adapt to a changing number of entities in the system.
  - *Flexibility.* The adaption to conditions which characterise the application during runtime.
  - *Stability.* The capability of a system to maintain its functionality even in the presence of frequent adaptions.

Tactics

Applying the methods for architecture selection as described in Chapter 4, this section identifies a set of tactics with the aim to ensure that the requirements of quality attributes are met. These tactics are used to identify *architectural patterns*, i.e. proven methods and best practices, to design a suitable architecture. Tactics also constitute the basis for the qualitative evaluation conducted in Chapter 7.

The following lists the relevant tactics according to [11] [103]. Tactics 1,2,3,15 relate to performance attributes, 5-11 help achieving modifiability, 9,10,11,15 support adaptability, 12-14 aim for security and 15 targets predictability.

1. *Reduce computational overhead.* Communication between entities is accomplished by using an efficient encoding.
2. *Manage event rate.* Entities try to actively control the event arrival rate for non-critical events.
3. *Introduce concurrency.* Messages are dispatched asynchronously and in parallel to other processing tasks, hence network bound delays do not affect internal operations.
4. *Fixed priority scheduling.* Events are assigned a priority according to their semantic importance, e.g. a fault message has a higher priority than a message transporting a general measured value.
5. *Maintain semantic coherence.* The goal of this tactic is to minimise dependencies on other modules and encapsulate dependencies within a module by choosing module responsibilities according to their semantic coherence.

Table 6.1: Remote backup protection quality attribute scenario

Scenario 1a: Remote Protection Function					
Source	Stimulus	Environment	Artefact	Response	Response Measure
Internal, external (third party device, auto configured)	Fault, Response	Normal, overloaded	Communication (IP stack, higher level protocol stacks e.g. IEC 61850), request dispatcher, query execution system, protection function	Protection function evaluation and initiate control command (operate breaker)	Time (clear fault before equipment damage), Time with varying number of entitites
Scenario 1b: Engineering and Maintenance					
Source	Stimulus	Environment	Artefact	Response	Response Measure
System operator	Modify, update, protection schemes	Runtime	Communication (routes), query execution system	Deployment of modification	Number of elements affected, effort, affects on quality attributes
Scenario 1c: Access Denial					
Source	Stimulus	Environment	Artefact	Response	Response Measure
Not trusted third party	Try to modify or update protection schemes	Networked	Communication, query execution system, communication and memory modules	Blocks access to modification service	Time to circumvent security measures. Service latency during an attack.

6. *Generalise the module.* Having a more general module allows for a broader applicability. For example, the query processor (s. Section 6.3) is a very general module which allows for flexible (re-)programming of the network.

7. *Information hiding.* This is a very general approach to decompose a system into sub systems and modules, assigning responsibilities to modules and specifying which information is public and which is private.

8. *Restrict communication paths.* Restricting the interaction of modules with each other reduces dependencies and prevents ripple events where modification of one module yields cascading modifications in many other modules.

9. *Use an intermediary.* If A depends on B in any other type than semantic, an intermediary C can be introduced which manages the dependency. Intermediaries can be repositories, e.g. blackboards, spaces, facades, proxies, mediators or factory patterns.

10. *Runtime registration.* A strategy to defer binding time to enable loose coupling of system components. Usually this tactic introduces additional overhead to registration and management of subscriptions. However, in static environments the overhead can be concentrated in an initial start-up phase.

11. *Virtualise the network.* In heterogeneous environments entities communicate using different protocols and standards. The strategy is to abstract from networked nodes by providing a unified access to services and data, e.g. using content based addressing.

12. *Authorise Users.* Authorisation assigns rights to users and groups of users such that they can access and modify data. Authorisation is realised using some kind of access control pattern (passwords, certificates, etc.).

13. *Maintain data confidentiality.* This tactic protects data from unauthorised access through, e.g. encryption of data and communication, Virtual Private Networks (VPN) or Secure Socket Layer (SSL) connections.

14. *Data integrity.* Critical data, such as fault information, needs to be delivered as intended. Hashes and checksums ensure that delivered data has not been tampered with.

15. *Virtualise the processing kernel.* Subdividing processes into discrete blocks of computation enables the achievement of deterministic real-time behaviour. Using event based kernels, however, may yield sub-optimal utilisation of hardware which restricts number and complexity of services executable on a device.

6.1.2 Scenario 2: Automated Outage Management

While the first scenario focussed on a field level mechanism, the scenario described now is located at the interface between the field or control level and the business process level. Since multiple players are involved in the scenario, the description starts with the introduction of roles and associated responsibilities.

Description

Figure 6.2 illustrates the typical roles and relationships in power networks. Generation companies (GenCos) operate power plants; they can be located at different levels of the physical energy network, ranging from supergrid (e.g., large coal power plants or nuclear plants) over high-voltage grid (e.g., industrial powerplants) to medium- and low-voltage grids (wind or solar parks down to private solar installations).

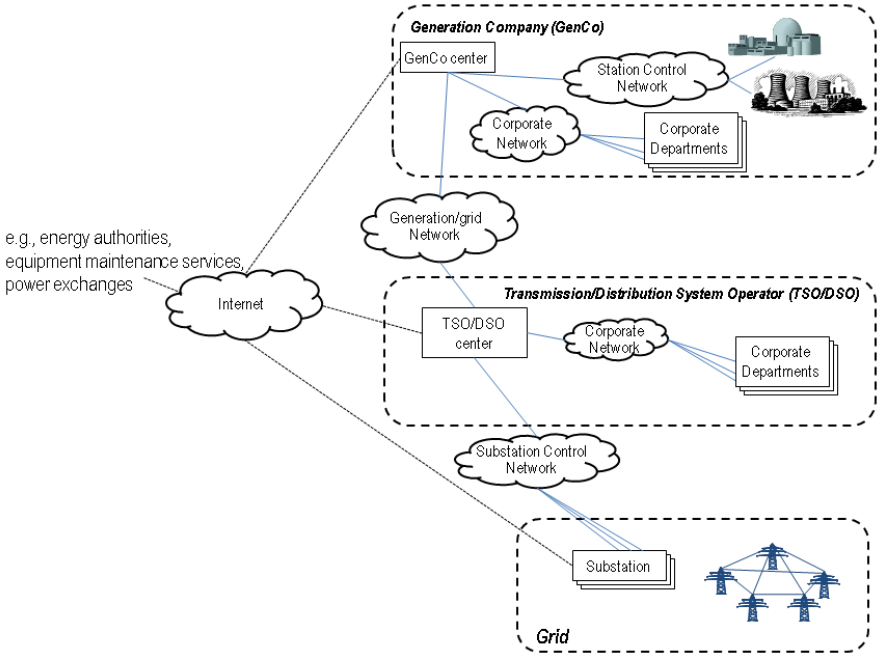


Figure 6.2: Power network topology (inspired by [28])

The control centre of a GenCo is responsible for co-ordinating two levels of activities, which in the sequel is referred to as *vertical integration* and *horizontal integration*. First, using supervisory control and data acquisition (SCADA) systems, the operating centre remotely controls the equipment located in the generation stations, with which it is normally connected via a dedicated plant control network to guarantee real-time control. These activities depend on field information and events coming from a multitude of sensors, devices, plants, and substations to be filtered, aggregated, and visualised. The process

of gathering, aggregating, routing, and interpreting this information within an enterprise control system is called *vertical integration*.

Second, the information gathered using vertical integration is used as a basis for planning, enacting, monitoring, and co-ordinating both internal and cross-organizational business processes. The business processes involving cross-organisational interaction and co-ordination with the partners in the energy network, and, in particular the transmission system operators (TSOs), and the distribution system operators (DSOs) are called *horizontal integration*. One example of this is the process of co-ordinating activities between a GenCo, TSOs, and DSOs in case of an outage. Vertical integration is used to determine type and extent of a local outage at the GenCo. In dealing with a local outage situation, the resulting information will support the decisions made in the business processes that guide co-ordination between the GenCo and its partners. Hence, the effects of the local outage can be isolated and spread and malfunction in other parts of the network can be prevented.

In general, the propagation of outage information involves three principal actors. The first actor is a system operator (TSOs/DSOs) who has a complete overview of the tie line maintenance and operation and is in a position to provide a coherent picture of the situation at a given instance in time. The second actor is a market information aggregator, who may be a commercial entity that simply specialises in providing electricity market information, and who offers the information to the public. Such a provider may also make the information available to a selected distribution list as an additional service. Finally, the third actor is an information receiver or interested market participant who wishes to obtain such information.

Already in 1999, the European Transmission Systems operators Organisation (ETSO) was founded to harmonise and develop the European electricity market. ETSO members include the Union for the Co-ordination of Transmission of Electricity (UCTE), the association of TSOs in Ireland (TSOI), the United Kingdom TSO association (UKTSOA), and the association of the nordic TSOs (NORDEL). ETSO takes care of cross TSO data exchange and standardisation. ETSO also standardises cross TSO processes like imbalance settlement, reserve resource planning, and outage transmission. In this context ETSO has standardised an electronic document that system operators may use to transmit information about outages to a market information aggregator. An outage document is issued by a system operator and refers to information for generation over 100 MW network lines which have influence on the offered capacity. Outage events can be either planned, i.e., planned shutdown of an asset, or unplanned, i.e., the forced shutdown due to failure or other emergency situations. In the current release, the outage document is limited to outages of tie lines and network interconnections. Future extension to other outage scenarios

is intended.

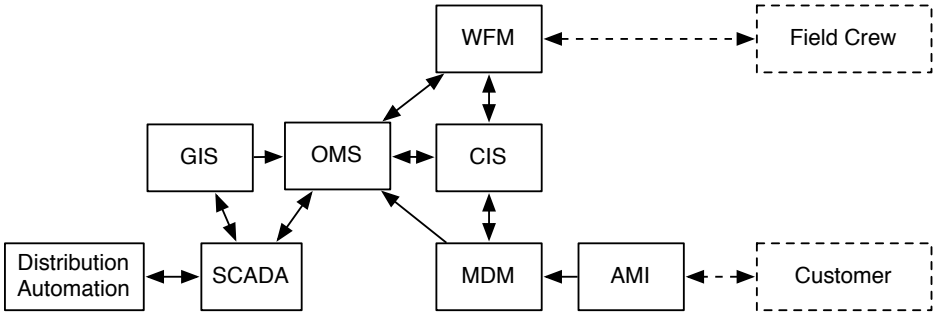


Figure 6.3: Outage management system integration (inspired by [71])

Outage Management Systems (OMS) are tightly integrated (Figure 6.3) into the utility's IT infrastructure. Recent deployments integrate various subsystems such as trouble call, Customer Information Systems (CIS), network connectivity, field data, Work Force Management systems (WFM) and geo-spatial information. In the near future, Smart Grid technologies such as Advanced Meter Infrastructure (AMI) including Meter Data Management (MDM) will further improve outage management system by providing real-time data on service availability and restoration.

Scenarios

The outage information publication process can be broken down to three scenarios, namely (i) outage detection, (ii) outage information publication, and (iii) outage information modification:

- *Outage detection (scenario 2a):* Detecting outages, especially in remote areas, is typically a manual process, i.e. customers recognising the outage, call their utilities to complain about it. In the US this fact causes average outage times (measured as System Average Interruption Duration Index (SAIDI) or customer minutes lost (CML)) of 120-160 minutes whereas it is 60-80 minutes in Europe. Quite obviously, automated outage detection is of immediate benefit for utility and customer. In order to fully automate outage management, the system needs to be able to detect outages and anomalies. Therefor, it must be able to monitor respective assets, access their status data and have means to identify the faulty component.

- *Outage Information Publication (scenario 2b):* whenever an outage situation occurs (either forced or planned) the information needs to be published. The conformity of publications is validated and if the information is incorrect the publication is rejected

and the incident is logged. If the information is correct it is stored persistently and accessible for interested internal as well as external partners.

- *Outage information modification (scenario 2c):* an outage situation may be modified to indicate its progress or to correct any data that is found to be invalid. Accordingly, the following possibilities exist:
  - For an outage the following information may be revised: start date, start time, end date, end time, and affected interconnector.
  - For an outage the following information may be added: start time, end date, end time, and affected interconnector.
  - If an affected asset is incorrect or the planned maintenance is cancelled prior to the start time, the outage in question has to be deleted and eventually new outage information has to be provided.

Quality Attributes

Table 6.2 summarises the scenarios elaborated in the previous paragraph using the method described in Chapter 4. From the description the following quality attributes are derived: observability, awareness, availability, integrability, adaptability.

- *Observability.* Individual entities may undergo a complex state evolution when executing processes in the system. Often entities must co-operate to achieve a common goal. Failure detection and state assessment of remote entities rely on open an interface to state and performance information.
- *Awareness.* Entities are highly connected. Actions taken by one entity may have effect on other entities. Complex cascading effects may influence system performance and stability. When making control decisions, entities must be aware of either the full system state or an estimation thereof.
- *Availability.* Industrial systems require individual entities to be reliable and highly available. The requirement includes the availability of function as well as data.
- *Integrability.* Outage detection and publication mechanisms directly interface with the business process infrastructure of the utility. Aiming fully automated processes, integration into the existing IT infrastructure is mandatory.
- *Adaptability.* Similarly to the previous scenario, adaptability in the sense of scalability, stability and flexibility plays an important role. Power system communication

infrastructures are underutilised during normal operation. In fault situations, however, multiple sensor equipment is triggered which causes intense data exchange between entities. Moreover, additional equipment is excited which further boosts the load on communication and compute infrastructures. Systems must cope with these varying loads and maintain functionality.

- *Testability.* In this scenario the system gains a new quality of complexity. Consisting of many modules that are distributed on a global scale, sub modules of the OMS must be testable to verify their correctness before being integrated.

Tactics

In the previous scenario description quality attributes and tactics have been identified which also apply for the outage management scenario. Integrability, observability and awareness are new attributes which can be achieved using the following additional tactic.

- *Language Interfaces.* Languages constitute a natural interface for communication not only for humans but also for technical systems. Instead of providing object based interfaces, coupling between components or services can be achieved by providing a language with shared vocabulary. Components provide a language interpreter with a respective grammar as their primary interface.

In the following sections the tactics identified in both scenarios are used as foundation to build an architecture. The discussion starts with a high level view on key components and, gradually, adds more detail when covering individual components and their relationships.

6.2 Architecture Overview

After introducing the key concepts of this work in Chapter 5, concretising the technical requirements as well as choosing appropriate tactics in the beginning of this chapter, the following sections present an architectural solution.

The information architecture integrates all assets, entities, business players, consumers and providers of energy services. It includes enterprise IT as well as the transmission and distribution network with components hosted by various devices such as energy automation equipment, meters, SCADA systems, sensors and further reaches into consumer homes on smart meters and other devices. All participants are networked by some kind of

Table 6.2: Vertical integration quality attribute scenario

Scenario 2a: Outage Information Creation					
Source	Stimulus	Environment	Artefact	Response	Response Measure
Internal	Fault or planned outage	Normal	query execution system, device state memory, monitoring function	Detection of the fault, generation of an outage document	Detection rate, Detection time
Scenario 2b: Outage Information Publication					
Source	Stimulus	Environment	Artefact	Response	Response Measure
System operator, outage detection system	Detected or planned outage	Networked (cross-organisational)	query execution system, index cloud, access control modules, notification system	Storage outage information and notify observers	Availability time, notification time
Scenario 2c: Modification of Outage Information					
Source	Stimulus	Environment	Artefact	Response	Response Measure
System operator	Update the outage information	Runtime	query execution system, access control module	consistent update of the data	Availability time, notification time

network technology i.e. ethernet, DSL, PLC or other. In the following, unless otherwise stated, participants are referred to as *nodes*.

The architecture links all nodes of the network in an application layer overlay to provide a global address space for all assets in the network. The overlay manages the mapping from virtual node addresses to physical hardware addresses and abstracts from network topologies and technology. The mapping supports both keywords as well as complex declarative queries. Keyword mappings are realised by a distributed hashtable (DHT) which is spanned over all networked devices (Figure 6.4). In order to support complex queries, a subsection of the address space is dedicated to the so called *index cloud*, a globally accessible meta data index and storage for data items. The cloud consists of index nodes organised in a dedicated structure as detailed in Section 6.4. Additionally to the central index, data sources host a local query engine which is capable of processing distributed queries as explained in Section 6.3. Index cloud and query processors implement a data-centric architecture. Data can be described, structured, distributed and processed using a data-centric programming language.

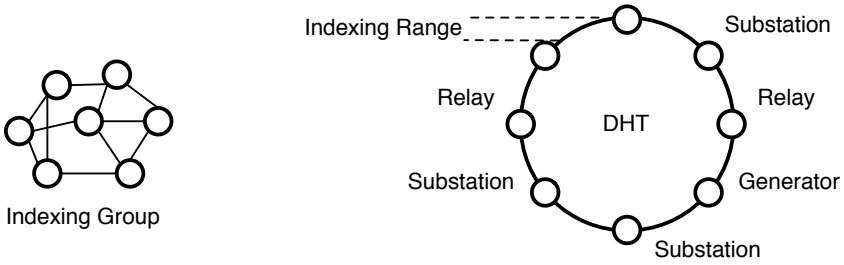


Figure 6.4: Overall architecture

Avoiding transferral of field data to a central service, the architecture can achieve higher efficiency and more detailed and frequent monitoring of the power grid by processing data close to its origin. Optimal operation would be achieved if all nodes are directly connected. Due to the diversity and age of the existing infrastructures this might not be realistic. Instead, integration methods are required that connect legacy devices to the ecosystem. The architecture supports three levels of integration: for data sources with sufficient compute resources, the query engine can be directly hosted by the data source. Data sources which do not provide the required resources to support a query engine by themselves but have a device in their proximity are integrated by the neighbouring query engine acting as a proxy that reads individual measurements from the device, and processes them in the query context. Finally, data sources neither support a common communication standard nor allow adaption of their software, require an additional hardware component which acts as gateway to read, process, and forward data.

Node meta data provides information on device capabilities and attributes along with a node locator. It must not be confused with data item meta data as introduced in Section 5.4. Node meta data is structured in a table called *nodes* which is stored in the memory of every node. Using the sparse table model, the nodes table contains an unlimited number of columns yet for each row two columns are mandatory namely: *ID* and *physical address* (Figure 6.5). The ID column equals the virtual address of the node which persists even if the physical address, which is the address supported by the underlying network technology, changes. This is particularly useful for nodes that frequently change their physical addresses, e.g. mobile nodes or nodes connected via DSL. By convention the physical address has the format *type://address*, e.g. ipv4://133.23.52.2 for IP based networks and mac:// 00:25:00:44:62:a5 for ethernet mac addresses.

ROWID	ID	Physical Address	...
5	43	ipv4://192.168.2.1	12hz
23	345634	mac:// 00:25:00:44:62:a5	
78	2345634	mac://00:25:00:59:2f:9b	50hz
...	9764	ipv6://::0ADE:70D6	

Figure 6.5: Nodes table

The nodes table allows for device lookup based on their descriptions (e.g., “send a message to a device which can measure frequency”). Additionally to discovering devices, the virtual addressing scheme can be used to route information in a declarative way (e.g., “send measurements from devices that can measure frequency to a device that can archive data”). To formally specify node discovery as well as routing information, the query language introduced in Section 6.5 is used.

An ID is assigned to every node during bootstrap. Therefor, nodes chose an arbitrary ID from a very large ID space. The size of the ID space guarantees that the ID is unique in the ecosystem. The ID uniquely identifies the node within the ecosystem. Along with the ID, private public key pair may be generated and can be used to sign transmissions between nodes. However, the signature is optional reflecting the fact that some nodes have severely constraint resources and hence are not capable of executing any cryptographic functionality.

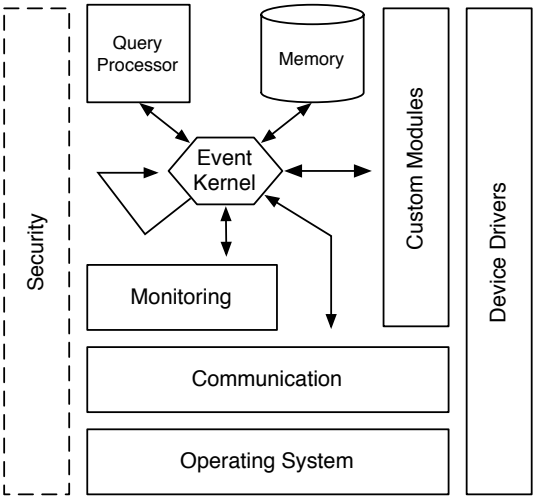


Figure 6.6: Node architecture

6.3 Node Architecture and Query Processors

A *node* is the central concept of a networked entity in the ecosystem. It is comprised of a set of standard modules yet can be extended by further device specific capabilities. The standard modules are (i) *Communication*, (ii) *Memory*, (iii) *Kernel*, (iv) *Query Processor*, (v) *Monitoring*. Security is not a module but a general concept which may be realised as part of other modules, e.g. communication, memory as well as the host operating system.

Figure 6.6 shows the five modules and their interaction. The kernel executes scheduled instructions, called *events*. Modules interact solely by scheduling events with the kernel which either rejects them or guarantees their timely execution. The following sections explain purpose and functions of each of the modules as well as their interactions between each other.

6.3.1 Communication

The communication module provides an abstraction of the physical network. Its interface allows to send and receive messages to either one or many other nodes whereby nodes can be addressed by their ID or a declarative query. Despite the system DHT, the module further allows to join and leave additional overlays. Messages contain an address header, ID or query, a sender address with optional signature, followed by a type field, a conversation id, content length and a content field of arbitrary length. Header fields are separated by a separator tag `#`. Message type 0 is reserved for the transmission of programs (s. section 6.3.3) while other types can be defined by the user. The conversation id enables

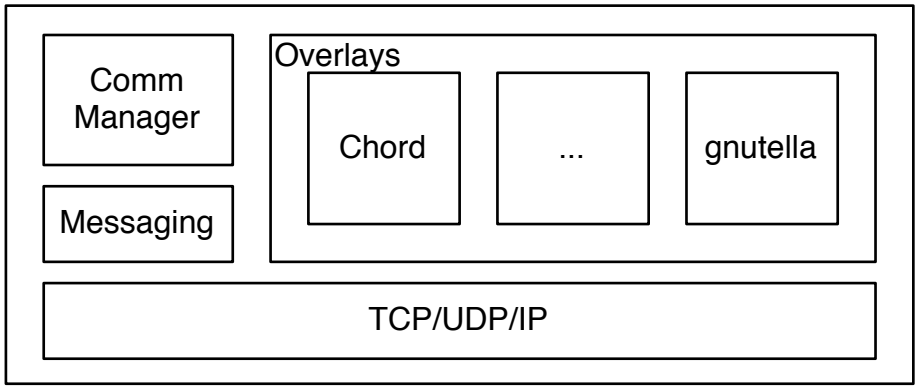


Figure 6.7: Communication module

the communication module to efficiently map different messages to the same interaction. The content of type other than 0 is opaque to the communication module and requires an application or program for interpretation. Type 0 messages are placed in the memory program queue (s. Section 6.3.2) and an event is scheduled with the kernel to initiate its execution (s. Sections 6.3.3 and 6.3.4).

Messages that expect a response are sent by specifying an response event. As soon as the response arrives, the communication event maps the response to the event by the conversation id. Subsequently, the event is scheduled for execution, e.g. a message to request a data item from another node would include an event to write the item to memory using a defined key.

The communication module maintains a table called *nodes* in the local memory. The table is a local version of the global *nodes* table and contains all nodes currently connected. It has one further mandatory column called *latency* which is updated by the monitoring module. New nodes are added to the table when the communication module establishes a connection to a previously unconnected node. Nodes are removed from the table if the connection is closed by the communication module, the monitoring module detected a failure or a node left the system.

To discover nodes and route information between them, the module maintains one or several overlay networks. While every node participates in the system wide DHT, for certain applications, e.g. to group nodes that are geographically close or nodes measuring a similar event, the communication module can create additional overlay networks. If a node participates in overlays other than the the system-wide DHT, additional address columns are inserted into the nodes tables. Accordingly, the overlay to be used to send a message is determined by the specified address which is in the format: *overlay://address*, e.g. *gossip://abc*. To send a message to other nodes their physical network addresses need

to be determined. If the specified address is an ID, the system-wide DHT is used to determine a route to the receiver. If the address is specified declaratively it is forwarded to the query processor which retrieves the information as described in Section 6.3.3.

Additionally to overlays, the communication module maintains local network structures, e.g. node groups or multicast trees. These structures are used to efficiently disseminate or collect information for specific tasks, e.g. high detail state assessment or load balancing. Local structures, however, are defined implicitly by rules. The communication module merely accomplishes discovery and maintains availability information. The concept of local structures is further detailed in Section 6.3.3.

Depending on the security mechanism chosen for the node, the communication module may be responsible to execute authorisation procedures. Depending on the domain and device type different security requirements may exist. Therefore, the communication module does not specify a particular security mechanism. However, an example of a fine-grained, role-based data access mechanism is provided in Section 6.6.3.

6.3.2 Memory

The memory module provides a mechanism to store and retrieve data items. Its interface includes `store<K,O>`, `get<K>` and `list<>` operations. Data items are stored and retrieved using an arbitrary key `K`, e.g. data item ID. By convention four keys are reserved namely: *event list*, *programs*, *nodes* and *state*. The event list contains the currently scheduled events (s. Section 6.3.4). The data item identified by *programs* contains a queue of programs currently running on the node. It is maintained by the kernel. The item *nodes* is a table containing all nodes currently known to the node. The data in the table is maintained by the communication component. The *state* refers to a table representing the current state of the physical or virtual device embodied by the node. Examples are the position of a switch or the voltage the device measures.

The list operation lists all objects currently stored in memory. It is used for maintenance purposes to check the validity of data items that are currently stored in memory. Expired items are removed from the memory and tables are administrated accordingly by removing pointers to expired rows. However, determinism for the clearance of expired data is not guaranteed. The only guarantee the memory assures is that the expired data is removed shortly after it expired whereby “shortly” is not further defined. Changes to the memory module become visible only between kernel time slots (s. Section 6.3.4). For instance, consider two programs A,B and one data item identified by key K. If both programs execute within the same time slot and program A, which executes first, changes K, the

changes are not visible to B until the next time slot. If B tries to change K within the same time slot an exception is raised and any changes previously committed in this time slot by B are rolled back.

Additionally to providing storage for data items, the memory acts also as cache during query processing. Storing intermediate query results in memory increases the performance especially for continuous queries as well as data items that are requested frequently. Assuming that data items, e.g. representing a measurement, are valid for a limited period of time, the query optimiser detailed in Section 6.3.3 can achieve considerable improvements by including the content already contained in memory.

The memory component is the only component allowed to have state. Other components must use the memory if they want to retain a value between computation cycles. Since operations in modules must not block, components push their current state to the memory if a computational step is complete and retrieve their state upon the next activation. This ensures that the entire node state is captured by the memory component. Using its interface state and state evolution can be observed and actions to stabilise, e.g. resource consumption or entropy reduction, can be taken (s. Section 7.3).

Similarly to the communication module, the memory module may need to execute certain security functionality. Besides authorisation, it may need to verify permissions on a per table or per item base. The security mechanism is most likely determined by a standard, e.g IEC, relevant to the application domain the node operates in. Therefore the specification of the memory module does not include a specific security mechanism. Instead, for an example of a fine-grained, role-based access control mechanism, it is refered to Section 6.6.3.

6.3.3 Query Processor

The query processor translates a query into a form which is executable on one or more nodes. Declarative query statements are compiled to one or many *programs* which are executed either locally or remotely. Using information from the nodes table, programs are optimised according to the quality of service constraints specified in the query. In the sequel, the concepts *program* and *optimiser* as well as *execution* are described in further detail.

Programs

A program is a sequence of asynchronously executing instructions. The instruction sequence is not static but can be rewritten during execution by adding or removing instructions. A program can hence reflect the current environment to react to failures. Additionally, a program can be split and distributed to more than one node. Programs can also join to execute on a single node.

The instruction set consists of *selectData*, *lookupNode*, *lookupTable*, *evalExpression*, *writeData*, *readData*, *assignQuery*, *destroyQuery*, *sendMessage*, *optimizeQuery* whereby:

- *selectData*: selects rows from a local table that match a given set of conditions
- *lookupNode*: uses *selectData* to retrieve nodes that match a given set of conditions
- *lookupTable*: queries local memory or global index to find a table with a given name
- *writeData*: writes a data item to memory
- *readData*: read a data item from memory
- *assignQuery*: assigns a query to a given node. If the assignment fails due to a communication error or exceeded resources, it triggers the optimise instruction with the aim of rewriting the query such that an alternative node is chosen.
- *destroyQuery*: stop a previously assigned query locally or on a remote node.
- *sendMessage*: send the specified message to a node identified by the given address.
- *optimizeQuery*: analyses the program in the context of the current execution environment, i.e. local memory and node tables. If more than one node is involved in the query, the optimise instruction tries to find the optimal node set to execute subqueries. Optimality is derived from the quality of service constrains specified with the query. For instance, an optimal node can be one with lower latency or a node with strong encryption capabilities. For each query context, the optimise instruction stores additional node information, e.g., overloaded or unreliable nodes, in memory.

All instructions work asynchronously using the memory to store intermediate results. Therefor, they allocate a private data item in memory referred to as *program context*. The program context stores the current status of the execution such that if an instruction is executed, the current state can be processed.

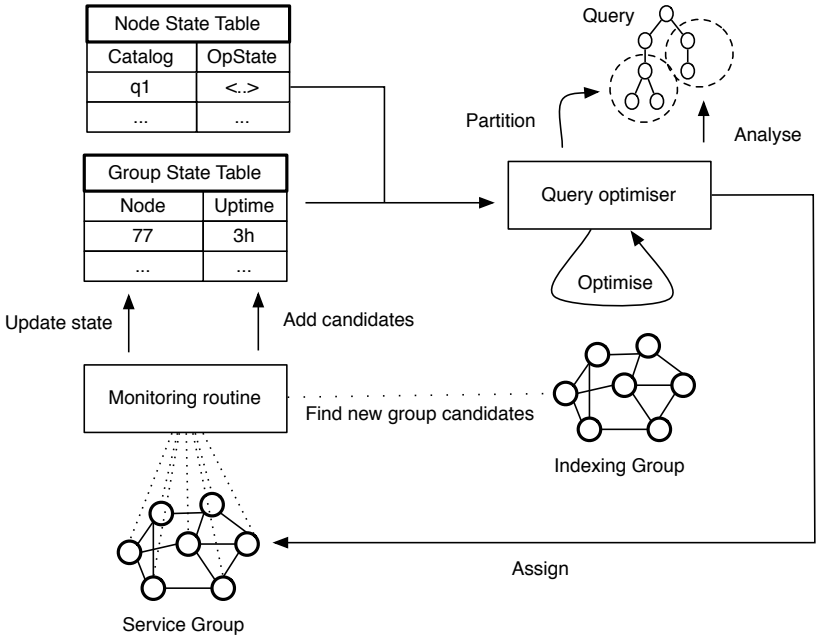


Figure 6.8: Program adaption during execution

Query Optimisation and Execution

The query execution subsystem is the most sophisticated of all modules. Its purpose is to analyse a query and create, given the information on the current state of the environment, an execution plan such that specified quality attributes are achieved. The section introduces the concept of the query optimiser as follows: first, key terminology is introduced. Second, using the key terms, the goal of the query optimiser is formally phrased. Third, key capabilities of the query processor are explained. Subsequently, the execution strategies are described and further illustrated by examples.

In order to provide a concise description the following key terminology is used.

- *Query q.* The query currently under investigation.
- *The set of nodes N.* The set of all nodes in the system.
- *Execution candidates* $n \in N$. A set of nodes, potentially capable to execute the query as a whole or parts of it.
- *Data sources s.* A set of nodes that cannot be replaced by other nodes, e.g. sensors. where $s \in n$.
- *Constraints c.* A set of criteria specifying the optimisation target.

- *Cost model.* A model to predict the cost of execution, given q, n, s and information on the current state of the environment.

The above concepts cover all relevant factors involved in the query execution and optimisation process. Using this key terminology, the query optimisation goals can be stated concisely.

> Goal: Given a query q, a set of nodes n, a set of sources s and a constraint set c, find a query structure such that the cost of execution becomes minimal under the constraints specified.

In order to be able to achieve the optimisation goals, the query processor module must, first, categorise queries according to their execution properties and scope, i.e. global or local. Second, it needs to apply a cost model to be able to differentiate between solutions in the solution space. The cost model includes the definition of a set of parameters and functions that quantitatively predict the cost of execution for a given query. Third, the query processor needs to implement an optimisation algorithm which, given q,n,s, and the information on the current state of the environment, searches the space of semantically correct alternatives to q with a view to minimise the cost of execution of q. Forth, the query processor must provide a runtime environment or control loop which continuously monitors the environment and progress of query execution. It analyses the monitored data for possible problems or better performing alternatives. The control may rewrite a query such that its execution costs are less than the current costs.

Queries can be classified into four categories: (i) *snapshot distributed*, (ii) *snapshot discovery*, (iii) *continuous distributed*, (iv) *continuous discovery*. Distributed queries are queries returning one or many result sets from one or more arbitrary nodes. Discovery queries are queries returning one or many result sets containing the meta data of a node from the index cloud. Snapshot queries are queries returning a single result set while continuous queries return a stream of result sets. A special form of continuous queries are subscriptions which return a result set only if it differs from the previously returned result set.

Discovery queries are not optimised other than checking if a matching result set is already contained in the local cache. Continuous discovery queries may be executed as subscription, i.e. they are registered at the index cloud and whenever an update occurs that matches[1] the query, a result set is forwarded to the receiver. Discovery queries are passed to the index cloud by choosing an arbitrary ID from the index range, looking up the corresponding index node by using the DHT and finally transferring the query to the

[1] Including the WINDOW constraint

index node for execution. For a detailed description of the execution process in the cloud including subscription processing see Section 6.4.3.

Depending on the number of nodes in the system, discovery queries may inflict considerable load on the index cloud. To assure that a continuous query includes all nodes for optimal result set generation, query processors need to contact the index cloud frequently looking for new execution candidates. Depending on the overlays supported by a node, the query load on the index cloud can be reduced. Therefor, a set of additional queries updates the local nodes table to contain all information necessary to restrict discovery to the local table only. Listing 6.1 shows such a set of queries, implementing a distributed clustering algorithm. Clusters are build up gradually by gossiping location information between nodes. At iteration $k = 0$, nodes are initialised with a random set of cluster centroids $\{C_{i,j,k=0} : 1 \leq j \leq K\}$ where K, the number of centroids, is a parameter to be specified. Each node determines its membership to the cluster with closest centroid. In each iteration, the ith node N_i picks a selection C of all n nodes at random and computes the current centroid by averaging over a determined value at the currently selected nodes. A node belongs to a certain cluster if its local value does not exceed a minimum distance from the average of all selected nodes. Nodes that share the same cluster are kept in the local nodes table. Nodes that belong to another cluster are purged from the local nodes table. The approach unloads the index cloud in two ways. First, the queries to discover new nodes are light weight with a small set of constraints. Second, once clusters converged, the discovery rate can be reduced allowing to include information from the nodes tables of other nodes in the cluster. Instead of the average function, an arbitrary correlation function can be used. Hence, nodes sharing a commonality form clusters and therefore decrease the effort necessary to execute and optimise distributed queries within the network.

While discovery queries are executed at the cloud, distributed query execution is handled locally by a continuous five stage process (Figure 6.9). In the first stage all information necessary for optimisation is collected. This includes look-up of data sources, retrieval of state and execution schedules from execution candidates, i.e. all known nodes capable of executing the query as a whole or parts of it. In the second stage the query is optimised and rewritten based on the information collected in stage one. The result of the second stage is a query plan, i.e. a program, containing all steps necessary to execute the query including placement of sub-queries on execution candidates. If no query plan can be found, e.g. because nodes cannot be resolved or due to conflicting constraints, the execution process terminates by skipping to state five. Otherwise, in the third stage, the query is rolled out to all execution candidates. If the rollout process fails, e.g. because one of the execution nodes failed or execution schedules changed such that the sub-query cannot be scheduled,

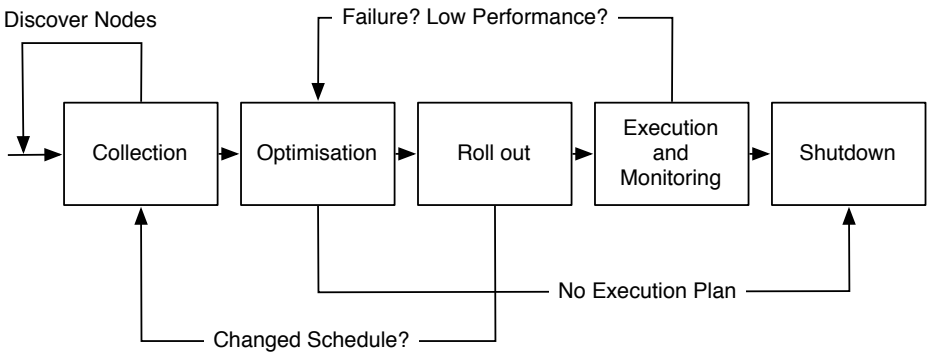


Figure 6.9: Five stage query execution process

the failure is recorded to the information collected in stage one and execution goes back to stage two. In the fourth state the query is executed and monitored. If query performance is not as expected or failures occur, the query is re-optimised by going to stage two. The fifth stage is reached once the query is complete or an unrecoverable failure occurred. In this stage unneeded information is purged and sub-queries are cancelled at the execution candidates.

The query execution process aims to find an optimal set of query configurations given the supplied constraint set. However, if followed strictly, query optimisation may destabilise the entire system or find only sub-optimal results as a large fraction of available resources are bound by query optimisation and re-allocations. As example consider two idle hosts and a query injected to be optimised for throughput (s. Cost Function A.3). The query will be scheduled at host 1. Depending on the induced load on the node, node 2 might a be better execution candidate causing re-configuration of the query and migration to node 2. Being at node 2 however, will make node 1 more attractive since its load decreased. If implemented poorly, the query might continue to "jump" from node 1 to another thereby generating considerable migration overhead. Hence advanced – entropy reducing – optimisation steps are supported (s. Section 7.3.2).

To illustrate query execution and optimisation, consider the continuous aggregate query Q in graphical form depicted in Figure 6.10. The query shall be executed on the example network shown in Figure 6.11. Informally, the query states the interest to retrieve the maximum of the values of the sensors 3, 4, and 5, and of the average of the measured value of sensor 1 and 2. The query does not contain any a priori knowledge about the network topology. All five sensors could be part of just one local network, e.g., within a substation; they could be connected each to a different network, or their connection could be a combination of the first two cases. The query does specify, however, a set C of quality of service constraints.

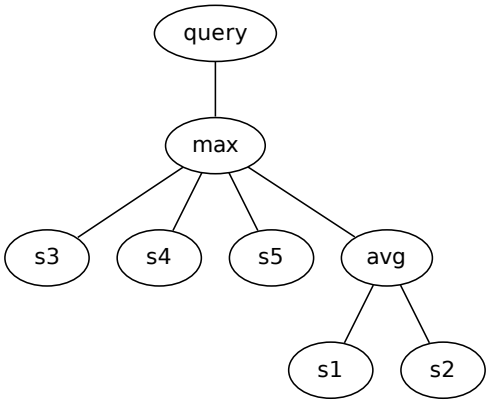


Figure 6.10: An example query

Further it is assumed that sensors S1-S5 are specified declaratively in the following way:

- Sensors 1-2 are to be chosen as of type "voltage sensor" and located in a specific area A_1, having the highest accuracy of all other voltage sensors in that area and featuring strong encryption.
- Sensors 3-5 are to be chosen as of type "voltage sensor" and located in area A_2.

The network contains in addition to the sensor nodes 1-5, nodes which are currently known to the query initiator, which is set to node 10. Nodes are connected via an IP network and there exist routes between each node. The edges of the network graph in Figure 6.11 show all one-hop links of individual nodes. Edges are labelled with the latency between the two nodes connected by the edge. Nodes have different compute capacities. Sensor nodes 1-5 have only minimal resources limited to sensing and delivering data. Node 10 has more compute resources, however its memory is limited to hold a maximum of one data item. Nodes 6-10 have enough resources to save multiple items as well as compute additional functions.

In the following, using an example network and query, the query execution process is illustrated. In stage one nodes are discovered by issuing look-up queries, containing the sensor description, to the index cloud. The discovery is the first step towards meeting the quality of service constraints as only nodes are selected that meet certain qualities, i.e. accuracy and security. Subsequently to discovery, the process advances to stage two where the optimiser needs to (i) rewrite the query such that it optimally fits the underlying network topology and (ii) find suitable nodes capable of executing sub-queries. Query rewriting starts by partitioning the query into all possible sub-queries. Thereby a sub-query is either

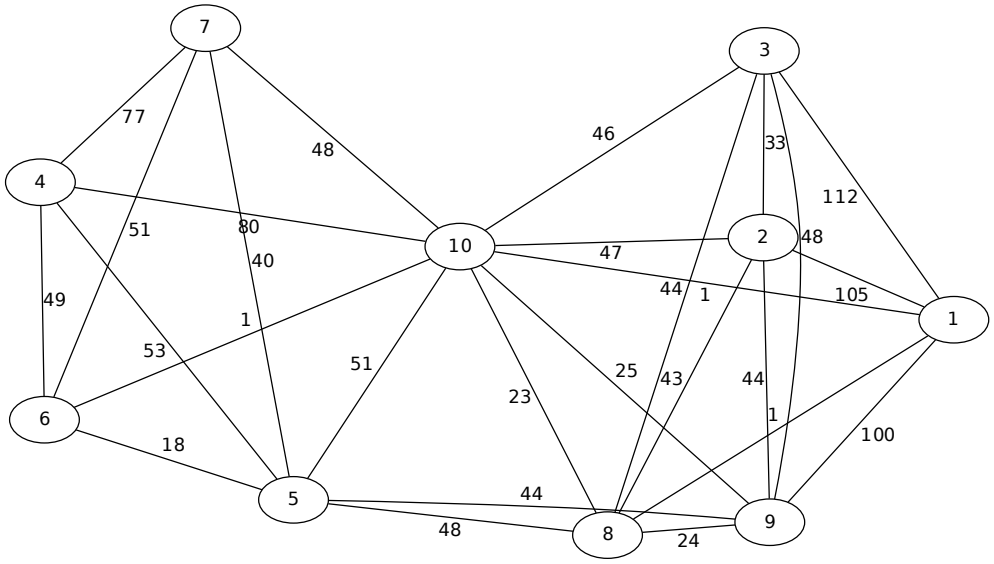


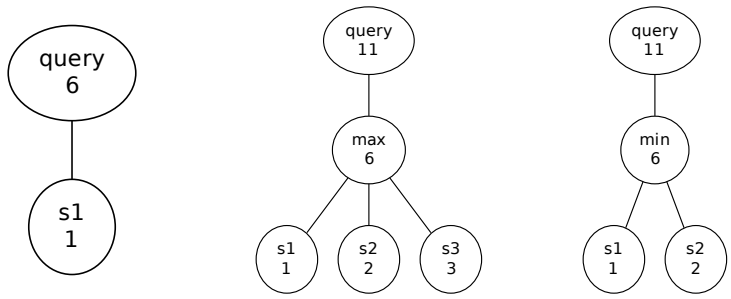
Figure 6.11: An example network

a query that retrieves a single data item from a node or an aggregation which takes two or more data items to merge them into a single item. Aggregations can be broken further into sub-queries until the minimum fan-in is reached, e.g. SUM(A,B,C,D)=SUM(SUM(A,B), SUM(C,D)). However, the semantics of the aggregation operator must be maintained, e.g. AVG(A,B,C,D,E) $\neq$ AVG(AVG(A,B,C), AVG(D, E)). In the context of the example query this yields: Q, Q_{s1}=S1, Q_{s2}=S2, Q_{s3}=S3, Q_{s4}=S4, Q_{s5}=S5, Q_1=AVG(Q_{s1},Q_{s2}), Q_2=MAX(Q_{s3},Q_{s4},Q_{s5}), Q_3=MAX(Q_{s4}, Q_{s5}), Q_4=MAX(Q_{s3}, Q_1), Q_5=MAX(Q_{s4}, Q_1), Q_6=MAX(Q_{s5}, Q_1), Q_7=MAX(Q_2, Q_1), Q_8=MAX(Q_1, Q_{s3}, Q_{s4}, Q_{s5}), Q_9=MAX(Q_1,Q_{s3},Q_{s4},Q_{s5}), Q_{10}=MAX(Q_4,Q_{s4},Q_{s5}),Q_{11}=MAX(Q_5,Q_{s3},Q_{s5}), Q_{12}=MAX(Q_6,Q_{s3},Q_{s4}).

Subsequently, after breaking Q into sub-queries, the optimiser builds all semantically correct variations of Q. Following the example, the optimiser would produce: Q, Q_{10}, Q_{11}, Q_{12}. Using the quality of service criteria, each query candidate is assigned a cost value according to Equation 6.1.

$$cost(Q) = \sum_i c_i \cdot w_i \tag{6.1}$$

Where $c_i = \sum_j \sum_l c_l(q_l)$ is the cost for the ith quality of service constraint over all sub-queries q_l and w_i is the corresponding weight. Cost functions for the different quality of service constraints, specified as detailed in Section 5.6, are listed in Appendix A.1 - A.5.



(a) Query A (b) Query B (c) Query C

Figure 6.12: Multi-query optimisation

For each query, a vector $\vec{w}$ of weights is generated using the constraint set C for the query. For example, a query that is to be executed as fast a possible would put a high weight on latency and service time and zero weight to all other constraints. The optimiser chooses the query such that $cost(Q) = min(\sum_i c_i \times w_i)$. To discuss the impact of QoS constrains on optimisation in the following, different constraint sets are evaluated. To be able to demonstrate individual influence of a constraint, only the corresponding weight is set to 1 while all others are set to 0. However, constraints regarding the resources detailed above remain considered. For a constraint set specifying a minimal latency for execution, the optimiser rewrites the query as depicted in Figure 6.14a. The optimiser did not rewrite the query. Figure 6.14b shows the placement of aggregation operators on the set of available nodes. The optimiser paid attention to the fact that node 10 is unable to store more than one item and thus placed all aggregations on node 9. In this example nodes 1-5 and node 10 cannot be used due to resource restrictions, leaving 8 and 9 for optimisation. Node 5 is connected to nodes 8 and 9 via a single hop. Nodes 1-3 are one hop away from 8 and 9 whereby the maximum latency to 8 is 44ms. Node 4 is two hops away from either 8 (4-5-8 102ms) or 9 (4-5-9, 97ms). Thus, since the latency is determined by the maximum latency between executing peers, node 9 is chosen.

The second constraint set in this example specifies to optimise for response time. Apart from latency, response time depends on computational speed of the node and the supported bandwidth. Figure 6.15 shows the result of the optimisation. Node 8 is more capable in terms of computation while the bandwidth supported by node 9 is larger. Hence, balancing between bandwidth and computational power, the optimiser places the average function on node 8 while putting the maximum with increased fan-in on node 9. Correspondingly, Figure 6.16 shows the result for a network composed of nodes with equal resources, minimal latencies and maximal throughput.

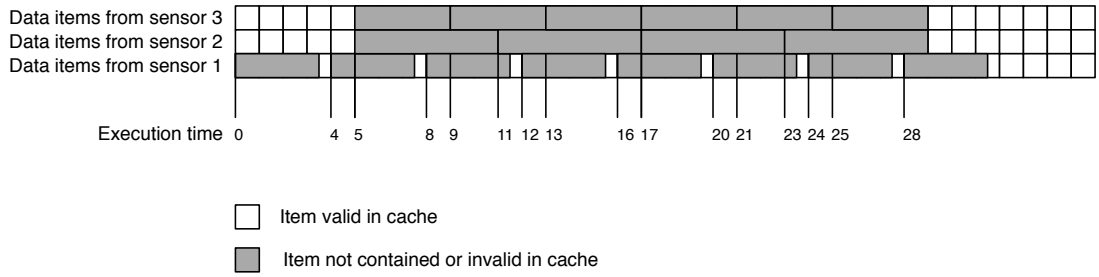


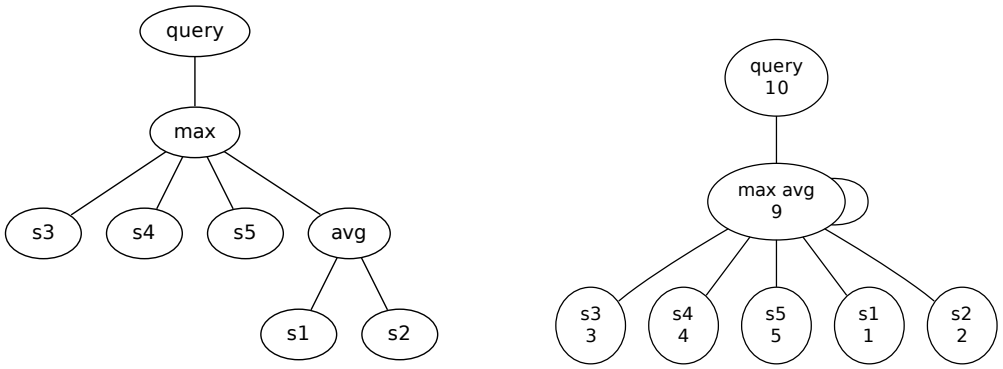
Figure 6.13: Valid data items contained in cache of node 6 during execution of queries A-C

Multi-Query Optimisation

Aside from static optimisation and assignment of queries to available resources, nodes optimise the combined execution of multiple queries. This section explains the process of multi-query optimisation and elaborates the difficulty associated with this step. Multi-query optimisation is also an important aspect covered in the evaluation conducted in Chapter 7.

During mutli-query optimisation result sets are shared among queries and multiple queries are consolidated into a minimum set of simplified queries. Hence, with multi-query optimisation, communication as well as computation costs can be reduced while still maintaining all quality of service constraints. While the general idea of reusing query results might seem very simple, in the following an example of three simple queries is provided, which demonstrates the complexity of this optimisation step.

Consider the placement of three example queries provided in Figure 6.12. Query A requests a measurement from sensor 1 every 3 minutes (Figure 6.13), query B requests the maximum of sensors 1-3 every 5 minutes (Figure 6.13) and query C requests the minimum of sensor 1 and 2 every 2 minutes (Figure 6.13). Furthermore, assume that readings of sensor 1 are valid for 3.5 minutes, sensor 2 for 6 minutes and sensor 3 for 4 minutes. All three queries are executed concurrently, but the execution of query B starts 5 minutes later than A. Query C starts 16 minutes later than A. All queries finish after 30 minutes (Figure 6.13). Obviously, all queries work on the same data items. When A and C start, they can reuse their results, hence they can be replaced by two queries α, β where α requests data items from sensor 1 every 3.5 minutes and β requests items from sensor 2 every 6 minutes. When query B starts, a new query γ is generated that requests items from sensor 3 every 4 minutes.



(a) Query (b) Placement

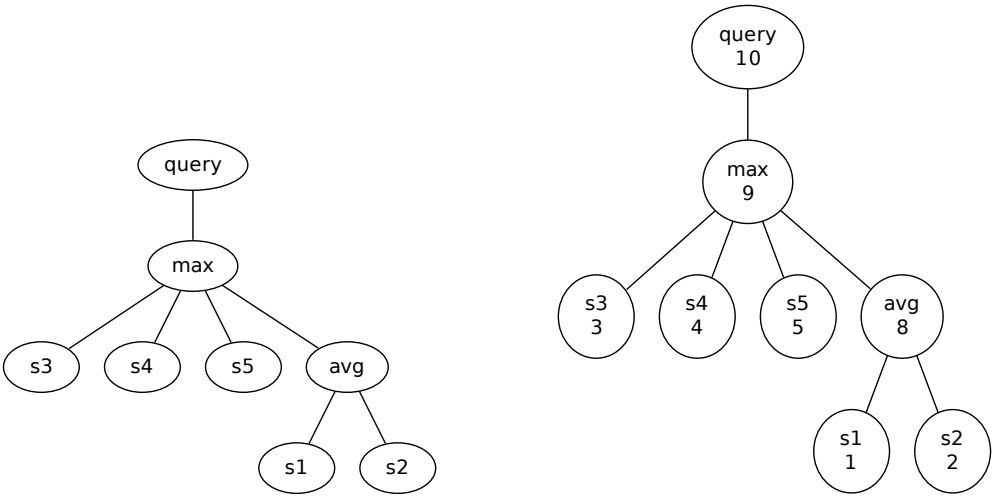
Figure 6.14: Optimisation for latency

Figure 6.13 shows the data item availability as requested during execution. Starting from the bottom, each row represents items from a sensor 1 to 3. Labels represent points in time when a query executes.

Instead of explicitly rewriting queries, multi-query optimisation operates twofold. Nodes provide a two way cache mechanism to reduce the number of required transmissions. First, received result sets and intermediate results are cached locally. Additionally, each node records sent items together with the send time. Therefore, a node knows which items are present at the receiver. Consequently, items are sent only if they have never been sent before, or if their validity in the receiver cache expired. Second, each query is optimised as elaborated in the previous section. In addition to the quality of service criteria, a cache criteria is used in the cost calculation, such that nodes that cache required items lead to lower costs of the placement. This approach has two benefits: first, there is no additional overhead of rewriting queries if new queries arrive or finish and, second, since items arrive only when they should, query windows do not need to be adjusted in the case items need to be retrieved acyclic. As elaborated in Section 7.3.2, aggregation, caching and multi-query optimisation may yield sub-optimal results. Using the caching criteria, however, continuous optimisation finds the optimal aggregation points and hence optimal multi-query execution.

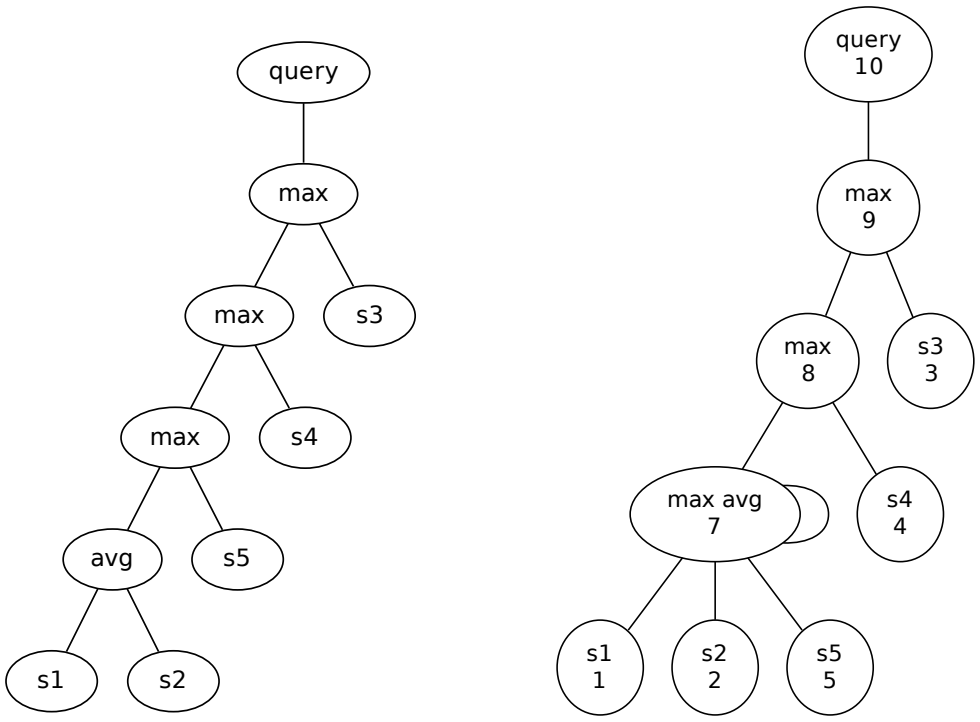
6.3.4 Event Kernel

The entire node is controlled by an event-based kernel module. The kernel operates by fetching an event from the event list (Figure 6.17) in memory and executing it. The event list is segmented into equally sized sections, called slots henceforth, each representing a



(a) Query (b) Placement

Figure 6.15: Optimisation for response time



(a) Query (b) Placement

Figure 6.16: Optimisation for throughput

discrete portion of time. Starting from the current instant, it extends into an (almost) arbitrary distant future. Therefore the kernel implements a local virtual clock c_l with the current time t_c being the current time slot. Events can be inserted into the list by providing (i) a timestamp specifying a point in time, relative to the current time, when the event shall be executed and (ii) the event code to execute. Insertion into the event list is constrained by the following:

- The timestamp must always lie in the future, i.e. $timestamp > t_c$ with t_c being the current time
- The number of possible events per slot is limited. If a slot has reached its capacity insertions are rejected causing program execution to fail.
- Events in the same time slot are executed in FIFO order with the exception of certain kernel events which are always executed first.

Event code must be linear and non-blocking without loops and threads. It may consist of program instructions (s. Section 6.3.3) or further kernel instructions. Kernel instructions are of high priority and control activity of all modules. The kernel instructions are:

- *fetchEvent*: a cyclic instruction scheduled as the first event during start-up. It fetches the next event in the event list and triggers its execution. After execution it schedules itself for $t_c + \Delta t$ whereby Δt refers to the slot duration. The instruction has the highest priority, and is thus executed first for each slot.
- *triggerCommunication*: a trigger for the communication module to send and retrieve messages or to join, leave or maintain overlays.
- *triggerMonitoring*: a trigger for the monitoring module to drive the monitoring protocol and update the local and global node tables.
- *triggerMemory*: a trigger for the memory module to drive maintenance tasks such as garbage collection.

How instructions are implemented depends on the underlying hardware and operating system. To prevent deadlocks, instructions must always execute asynchronously. If hard-real time is the goal, instructions must be implemented such that their execution stays within a strict time frame. This can be achieved by taking care that instructions, which could potentially require more time depending on the environment, are broken down into smaller instructions, e.g. memory maintenance may need more time for memories with

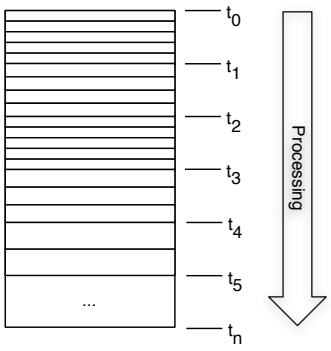


Figure 6.17: Event list processing model

many items than with fewer items, hence the maintenance instruction could be broken down to first work on the first k items, then on the 2nd, 3rd etc. items.

For certain situations, e.g. consistent read and write operations (s. Section 6.4.2), the synchronisation of event execution on more than one node is required. The kernel features synchronisation based on a variant of a *Chandry-Misra-Bryant* (CMB) protocol [95]. The method synchronises the local clocks of each node in a group of nodes G_s. A kernel may not proceed to next time slot until all other kernels within G_s acknowledged the complete procession of the current time slot t_c. To prevent deadlocks kernels exchange so called *null* messages to signal the group G_s that all events in the local timeslot t_c were executed. The constraint formulated earlier that no event may be scheduled with $timestamp <= t_c$ ensures that causality of execution is always maintained. In order to reduce the number of *null* messages, the kernel extends the original CMB protocol by *look-a-head* mechanisms [22], [86] [33] (Algorithm 1). The look-a-head mechanism exploits the fact that there will always be a network based latency between communicating kernels. Therefore, for each received message a small delay d_l is added prior to scheduling. The value of d_l is updated for each timeslot and set to the minimum of latencies of all nodes within G_s. The mechanism allows kernel to process all events in the event list until $t_c + d_l$ without synchronisation via *null* messages.

Algorithm 1 presents the scheduling and synchronisation methods. The main loop (lines 1-13) executes all events up to the execution limit determined by the minimum latency to kernels in G_s. Lines 14-21 adjust the minimal delay d_l. Line 23 sends a response to a kernel that reached its execution limit. The message contains the current local time - 1, to signal the last timeslot were all events have been processed. This is mandatory to ensure consistency if $c_{li} = c_{ll}$, i.e. current local time for kernel k_i equals current local time of kernel k_l with $k_i, k_l \in G_s$. Line 26 ensures that execution is not blocked by failed nodes (s. Section 6.3.5).

Algorithm 1 Kernel scheduling with look-a-head synchronisation

Require: For all kernels, $k_i \in G_s$, start time t_s is synchronised

```
1: while node = alive do
2:     if t_c < executionLimit then
3:         events = fetch(t_c)
4:         for all events do
5:             execute(event)
6:         end for
7:     else
8:         nodes ← MIN(k_l, nodeLatencies)        ▷ Get all nodes that block execution
9:         for all nodes, n_i do
10:            SEND(n_i, null, c_l)
11:        end for
12:    end if
13: end while
14: on receive message (MESSAGE, local time of sender c_ls) from k_i
15:    t_e ← t_c + d_l                            ▷ New local execution time t_e
16:    nodeLatencies ← nodeLatencies ∪ k_i, t_e
17:    if t_c − c_ls < d_l then
18:        d_l ← t_c − c_ls
19:    end if
20:    exeuctionLimit ← MIN(t_e, nodeLatencies)
21: end
22: on receive message (null, local time of sender c_ls) from k_i
23:    SEND(k_i, c_l − 1)                         ▷ Only events in c_l − 1 are fully processed
24: end
25: on receive message (NODEFAILURE, k_i)
26:    REMOVE(k_i,nodeLatencies)
27: end
```

6.3.5 Monitoring

Awareness of the state and availability of the other nodes is not only mandatory for query execution but also contributes to the robustness and efficiency of the entire system. Hence each node has a configurable monitoring module to collect information about other nodes as well as to provide a flexible interface for other nodes to gain information on its current state. Nodes monitor all other nodes currently contained in the local nodes table. The monitor module engages in a simple probing protocol by sending the monitored node a probe message. The monitored node responds with its state, meta data and failure hooks. The local nodes table of the probing node is updated with state and meta data information. A failure hook is a program to be executed if the monitored node failed, e.g. to remove the failed nodes meta data from the global index. A failing node is detected if more than three probe messages remain without response.

6.4 Index Cloud

While the previous sections introduced the modular node architecture, this section details the second key concept of the data-centric architecture for large-scale industrial systems, namely the *Index Cloud*. The responsibility of the index cloud is to provide global access to distributed information. Nodes publish a description of their capabilities (node meta data) and the data they provide (data meta data) in the index cloud and query the index cloud to find other nodes or data. Technically the index cloud is a storage for data items, however, it is not intended for massive volumes of data. High volume data should be handled at the nodes level as explained in Section 6.3.3. Aiming at a scalable meta data index for node meta data, the index cloud is optimised for the management of lightweight data.

The index cloud is comprised of a set of nodes, called index nodes henceforth, located in a specific ID range of the underlying DHT. To query the index, nodes choose an ID that lies within the index range at random and use the system DHT to route to the corresponding index node. Depending on the mode of operation, the type of consistency and level of availability, the ID range is segmented into several zones whereby each zone guarantees certain qualities of service.

The cloud has two modes of operation called *fully replicated* and *partitioned*. In fully replicated mode, all data published to the cloud is replicated to all index nodes. This mode ensures high availability and strong read performance. However, write operations suffer increased latencies and the cloud does not scale with the number of nodes. Additionally,

in the case of large data volumes, nodes may become unable to store the data or keep the index in memory. Therefore, in partitioned mode, all data published to the cloud is partitioned among a number of nodes. In this mode the index cloud consists of a number of node groups whereas each group contains the full index. Subsequent sections illustrate the self-organising formation of the index cloud with respect to both modes and their implications on data management and consistency.

6.4.1 Formation

The cloud is comprised of transient nodes such that no assumptions on their capabilities can be made. With increasing number of nodes in the cloud the probability of failure increases proportionally. Foundation for reliable operation of the index cloud is a stable structure that connects all nodes within the cloud. This section introduces structure generating and maintaining algorithms.

Fully Replicated

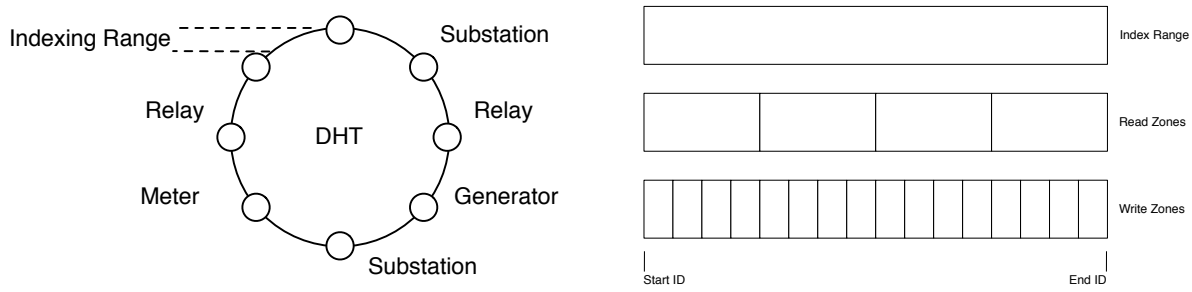
To simplify cloud structure maintenance, a dedicated node, the cloud master, co-ordinates all administrative tasks. As convention, the master is the index node with the lowest ID in the index interval. In theory any node could adopt the role of the master. However, as a reliable master can bring major performance improvements it is assumed that a capable and reliable node is positioned as master.

Algorithm 2 Formation algorithm for the fully replicated mode

Require: $\nexists$ Node n_i in list of active nodes

1: **on receive message** (JOIN, id_{cand}) from n_i
2: $n_{id} \leftarrow rand(range_{index})$
3: $activeNodes \leftarrow activeNodes \cup n_{id}$
4: $send(node_{id}, n_{id})$
5: **for all** $n \in activeNodes$ **do**
6: send(n, activeNodes)
7: **end for**
8: **end**

Index nodes join the DHT with an initial ID, ID_{cand}, chosen from a dedicated ID range, and query the master to assign them a suitable operative ID by choosing an unassigned ID from the index range with uniform probability (Algorithm 2). The master maintains a list of all currently active nodes in the cloud. Upon update of the list, it is pushed to all



(a) DHT (b) Read/Write Zones

Figure 6.18: Partitioned cloud zones

nodes in the cloud using the write operation detailed in Section 6.4.2. Additionally to the list, new nodes receive the full index from the master. The master also probes the nodes in the cloud by passing them a validity token. If the token is not acknowledged timely, the master removes the respective nodes from the list of index nodes. If a token has not been renewed and expired, nodes leave the cloud and the DHT as the connection to the master might be interrupted. They may try to rejoin the cloud at a later point in time.

The availability of the master is ensured by a watch dog mechanism. The nodes with the second and third lowest ID monitor the master by frequently probing the master by sending a validity token and requesting a copy of the current list of index nodes. The index node with the second lowest ID recognises a master failure by a response delay to the probe request or the arrival of coordination requests (joins) from other index nodes as they assume the master to be the node with the lowest ID. A master recognises its failures or separation from the cloud by an expiring validity token.

Partitioned

For the partitioned mode of operation, the index cloud is partitioned into so called *zones*. Equally to the fully replicated mode, a master co-coordinates the cloud formation. After joining, the master determines an ID for the new node. Therefor, it divides the index range into sections of equal size. By choosing an ID from the interval with uniform probability, the new node is mapped to a section. The node with the lowest ID in a section becomes zone master co-ordinating all administrative tasks of the zone. Zones at this level are called *read zones*. A read zone consists of k *write zones*, i.e. sub-sections of equal size within the read zone (Figure 6.18b). Again, a write zone master is the node with the lowest ID in the write zones range.

Whereas each read zone is a replica of the entire data currently stored in the cloud, write zones store only a subset of the data. Data within a read zone is partitioned by

the item identifier. Given a uniform distribution of identifiers, the following hash function uniformly maps objects with identifier OID to write zones:

$$H(OID) = \frac{ID_{zonemaster} + L_{readzone}}{(Hash(OID) \bmod S_{wz})} \tag{6.2}$$

where $L_{readzone}$ is the length of the read zone, i.e. $\|ID_{highest} - ID_{lowest}\|$ and S_{wz}, the size, i.e. the number of write zones, of the read zone and $Hash$ is a general purpose hash function such as SHA1. Since partitioning is at the item level and item content is opaque to the architecture it is the responsibility of the application to partition large items.

To read data from a partitioned cloud, client nodes choose an ID from the index range at random. The receiving index node distributes the query to all other write zones in its read zones as well as executing the query locally. Subsequently, the result of local and sub-queries is passed to the client. To write data to a partitioned cloud, clients choose an ID from the index range at random to forward the write request to the respective index node. Using Equation 6.2, it determines the write zone for the item to be written and forwards the write request. Additionally, it forwards the request to all other read zones using Equation 6.2 and adding the read zone length modulo the read zone length.

With increasing data volume or changed load patterns, zones can be adjusted such that data is distributed to more nodes. To join two bordering zones, the zone master with lowest ID of both zones acts as co-ordinator. To ensure consistency for data items with conflicting versions, it retrieves the item from n other read zones choosing the item with the newest version.

Similarly to the fully replicated mode, zone masters are monitored by the two nodes with the third and second lowest IDs. Additionally data can be replicated using the mechanisms described in Section 6.4.2.

While in the fully replicated scenario, the master merely assigns IDs to newly joining nodes, in the partitioned case, additional configuration overhead is needed. Read and write zones must be configured in dependence of the expected total read- and write load. If load patterns are known a priori and are expected to be stable, this process can be done manually during design time. However, using the monitoring capabilities of the nodes as illustrated in Section 6.3.5 and feeding this information back to the index master, the cloud can be reconfigured automatically. Since an index node's performance depends on the hardware, available resources, network infrastructure, caching mechanism, write-read ratio and number of items stored, a prerequisite of the cloud self-configuration or self-optimisation is a model of the throughput per index node. Particularly the relationship between read and write operations has influence on the performance. Replicating a data item increases read performance proportionally with the number of replicas. However,

Algorithm 3 Formation algorithm for the partitioned mode

Require: $\nexists$ Node n_i in list of active nodes

1: **on receive message** (JOIN, id_{cand}) from n_i
2: $n_{id} \leftarrow rand(range_{index})$
3: $activeNodes \leftarrow activeNodes \cup n_{id}$
4: $master_{write} \leftarrow$ GETWRITEMASTERFORID(n_{id})
▷ Update performance model
5: $\vec{m} \leftarrow$ PROBE($n_i, loadPattern$) ▷ Probe with current load pattern
6: $nodeModel \leftarrow$ APPROXIMATE($\vec{m}$) ▷ Levenberg-Marquardt [113]
7: $cloudModel \leftarrow cloudModel \cup nodeModel$

8: SEND($node_{id}, n_{id}, master_{write}$) ▷ Inform new node of write master for registration
9: OPTIMISE(cloudModel, loadPattern) ▷ Optimisation may be done lazy
10: **end**
11: **function** OPTIMISE(model m, pattern p)
12: $\vec{c_{opt}} \leftarrow \vec{c_{current}}$
13: **for all** Configurations $\vec{c_i} \in \vec{C}$ **do**
14: $t \leftarrow t +$ EVAL($\vec{c_i}, m, p$)) ▷ Compute throughput based on $\vec{c_i}, m, p$
15: **if** $t > t_{max}$ **then**
16: $t_{max} \leftarrow t$
17: $\vec{c_{opt}} \leftarrow \vec{c}$
18: **end if**
19: **end for**
20: **if** $\vec{c_{opt}} \neq \vec{c_{current}}$ **then**
21: $\vec{c_{current}} \leftarrow \vec{c_{opt}}$
22: PARTITION($\vec{c_{current}}$)
23: **end if**
24: **end function**
25: **function** PARTITION(Configuration $\vec{c}$)
26: **while** $i \leftarrow i + |activeNodes|$ **mod** $c(readers) < index$ **do**
27: $master_{read} \leftarrow i$
28: $j \leftarrow i$
29: SEND(ASSIGN_READ_MASTER_ROLE, i, $master_{cloud}$)
30: **while** $j \leftarrow j + |activeNodes|$ **mod** $c(writers) < i + |activeNodes|$ **do**
31: SEND(ASSIGN_WRITE_MASTER_ROLE, j, $master_{read}$)
32: **end while**
33: **end while**
34: **end function**

maintaining consistency causes write operations to become more resource demanding in a replicated scenario. Similarly write performance can be increased by partitioning data over many nodes. In turn, read performance decreases since many nodes must be contacted to execute a query. Write-read ratio has also influence on the performance of caching. Low write-read ratios allow for increased throughput by caching while many writes disable caching benefits. Using this information, the master creates a structure model of the cloud by collecting from each node measures on their read and write throughput in dependence of the write read ratio (current load). Subsequently, zones are configured, i.e. segmentation into a certain number of read and corresponding write zones, such that the total throughput can be guaranteed. For example, given an index node (implementation and hardware) with an average throughput[2] of 1986 read and 14 write operations per second, a write-read ratio of 0.007 and a total required throughput of 16000 read and 115 write operations per second, a cloud with 6 read zones and 41 write zones would meet the requirement. For a detailed discussion on finding the optimal configuration refer to Section 7.4.1.

Algorithm 3 presents the formation mechanism. When sending a join request the new node is probed with a load pattern characteristic to the current load pattern. The measurement yields an updated performance model of the index cloud. The new node receives an id and corresponding write master. Subsequently it contacts the write master for registration (Algorithm 2). Optimisation and partitioning may be done after each node join. However, for large index clouds it might be more efficient to initiate optimisation after a series of joins and leaves.

6.4.2 Index Data Management

Being able to scale to hundreds or thousands of index nodes, the index cloud is a data store providing the illusion of infinite resources. While formation algorithms explained in the previous section maintain the structure of the cloud, this section details the management of data within the cloud by describing the algorithms and concepts for storing and retrieval of data under a particular quality of service constraint, namely: *consistency*.

In [83] Leslie Lamport defined three consistency models for data stores: *safe*, *regular* and *atomic*. Safe consistency means that a read which is not concurrent with a write returns the last value written. Regular consistency guarantees that a read always returns a written value which is not older than the value written by the last preceding write. Finally, atomicity guarantees regularity plus it ensures that a read does not return an

[2]Numbers taken from an analysis of the load patterns on Siemens internal Web 2.0 applications

older value than a previous read. While atomicity is desirable for critical applications, it causes considerable overhead to enforce. On the contrary, safe and regular consistency are easier to achieve yet they are insufficient in distributed systems since in the first case a read concurrent with a write returns an arbitrary value and in the second multiple reads may return outdated versions.

Considering the fact that the index cloud is built of a large number of individual nodes based on commodity hardware, consistent management of data within the cloud is challenging. The asynchronous nature of node interaction as well as a variety of failure scenarios require sophisticated algorithms to consistently read and write data [21] [10]. Nodes of the index cloud are virtually close, i.e. within an ID range. However, in the physical world they can be separated over large distances, connected via heterogeneous networks and based on different hardware thus experiencing unpredictable delays. Consequently, depending on individual node dynamics, operations may take varying times to complete. Therefore, it is impossible to differentiate a faulty from a slow process. The definition of operation timeouts is sub-optimal since when chosen to strict, operations might fail in an overload scenario while when chosen conservatively, performance is poor.

In both fully replicated as well as partitioned mode, nodes of the index cloud must access multiple other nodes for read and write operations. Since no assumptions can be made on the reliability of nodes and their communication resilience, read and write algorithms must be able to compensate failure events. Additional to node and communication failures, Byzantine faults may yield corrupted data, e.g. due to hardware defects or security breaches.

Cloud data management features different modes of consistency to balance advantages and drawbacks of operation performance and consistency. In *strong consistency* mode, it is assured that if process A has made an update, subsequent accesses by A or any other process (B,C,D) will return the updated value. Strong consistency induces heavy write overhead on the system as all replicas need to be contacted and need to confirm that the value has been updated before the operation ends. In dynamic environments, e.g. in large replication groups, strong consistency may be unachievable as at any point in time a number of replicas may be offline. In *eventual consistency* mode it is not assured that all subsequent accesses return the updated value. However, if no updates are made, eventually all accesses will return the updated value. In the absence of failures the window of inconsistency is determined by communication latency, number of replicas and system load. Corresponding to the consistency modes, the data cloud features a *strong write* and *eventual write* method. Consistent write methods, however, are not sufficient. If process A writes to the cloud, aiming to write at n replicas and receiving n acknowledges, yet k out of n replicas write the item incorrectly or not all due to a hardware failure or malicious

intent, consistency compromised. Therefore, in addition to the modes for updating data, two read modes are supported. In *weak read* mode a value is read from a single node in the cloud. This mode is very fast yet susceptible for inconsistent data or maliciously or otherwise corrupted data. In *strong read*, data is read from n replica nodes. A value is returned based on a quorum decision of all nodes participating in the read. The mode is considerably slower than weak reads, however, it allows for consistent and save operations. In the following, the four operations are described in detail.

For a strong write operation, the client node selects an arbitrary index node from the index range. If the cloud is in partitioned mode, and the selected index node is not responsible for the item to be written, the request is forwarded to the appropriate write zone. The item is then written locally as well as to all replicas. In fully replicated mode these are all index nodes. In partitioned mode they are all nodes in the write zone plus all nodes in all write zones of other read zones. The write operation completes if respective nodes acknowledged the write. It fails if at least one node fails to acknowledge timely.

Similarly to the strong write operation, for eventual writes, an index node is selected at random. If the index node is not part of the correct write zone and the cloud is in partitioned mode, the request is forwarded. In fully replicated mode, the item is written to all index nodes. If k out of n index nodes acknowledged the write, the operation ends successfully and fails if less k node acknowledge timely. In partitioned mode, the item is written locally as well as to the write zone's masters of all other read zones. If k write zone masters out of n read zones acknowledged the write, the operation terminates successfully and fails otherwise. In both cases, replicas that failed to respond to the write request are recorded with the stored item at the node servicing the client request as well as the master. The node retries to write the item to the failed nodes until it discovers, by receiving a updated cloud list from the master, that the target node left the cloud or a new version of the item appears or the write completes successfully. If the node originally servicing the client request fails, the master assigns another node to retry the writes until the above stop criteria are met. Additionally, epidemic entropy reduction algorithms may be applied to speed up the harmonisation of item versions.

For a weak read operation, a client node selects an index node at random. In fully replicated mode, the index node executes the read locally and returns the result set(s). In partitioned mode, the read is executed locally as well as distributed to all write zones of the current read zone. If all write zones return their result sets, in case of a continuous read their result sets for a given iteration, the node receiving the client request joins the result sets and forwards them to the client. If at least one of the write zones fails to respond timely the read operation fails.

In fully replicated clouds, for consistent writes, the number of replicas that acknowledged a write equals the number of index nodes, hence a read on a single node guarantees strong consistency. However, unless explicitly stored with a data item, the cloud cannot determine whether the item was written strong or eventually consistent. Hence, for a read at least $n - w + 1$ nodes, where n the number of index nodes and w the number of nodes that need to acknowledge an eventual write, need to be contacted. Out of the $n - w$ received results, the result set with the highest version is selected and forwarded to the client. To ensure atomicity the selected version is written back to all replicas. In partitioned mode, all write zones in all read zones are queried and the selected result is written to the local write zone which initiates the write to all respective write zones in all other read zones.

6.4.3 Query Execution

Query execution in the index cloud differs from execution on regular nodes. Queries arriving at the index cloud are executed without prior data discovery. In the fully replicated cloud, the query can be directly executed on the local memory. Depending on the consistency requirement specified with the query, other index nodes must eventually be contacted before the result set is returned to the client. In partitioned mode, the query is executed at least locally and on all other write zones of the current read zones. In case of strong consistency other read zones are contacted as described in the above section before the result is forwarded to the client. Unlike to distributed execution, sub queries are not scheduled at other nodes for continuous execution but the index node which received the client request co-ordinates the execution by reissuing requests when needed. This eliminates the need to optimisation and reduces the overhead during reconfigurations and node failures.

The above sections detailed the second key concept of the data-centric architecture for large-scale industrial systems, the index cloud. The third important concept is a data-centric language which combines individual nodes and provides means to formally state data structures and data flow, quality attributes and application requirements. The query language is introduced in the following section.

Listing 6.1: Distributed discovery

```
CREATE TABLE CENTROIDS(DOUBLE c)

SELECT INIT()

N -> SELECT     *
     FROM       @NODES LIMIT 10 ORDER BY RANDOM
     WINDOW     (0, FOREVER, 10 minutes)

INSERT INTO N
    (SELECT * FROM NODES)
    WINDOW(1,FOREVER,10 minutes)

AVG -> SELECT   AVG(value)
       FROM     NODES
       WINDOW   (2, FOREVER, 10 minutes)

SELECT     CENTROID(c)
FROM       CENTROIDS
WINDOW     (4, FOREVER, 10 minutes)

DELETE FROM     NODES
WHERE           (value - MYCENTROID < 10)
WINDOW          (3,FOREVER,10 minutes)

FUNCTION INIT() {
   FOR (1 to K) {
       INSERT INTO CENTROIDS(RANDOM)
   }
}

FUNCTION CENTROID(centroid) {
    DOUBLE minimum = 100;

    FOR EACH (centroid) {
        IF (centroid - AVG < minimum) {
            minimum = centroid - AVG;
            MYCENTROID -> CENTROID
        }
    }

    UPDATE CENTROIDS
        SET    C=AVG
        WHERE C=MYCENTROID
}
```

6.5 Query Language for Service Ecosystems

Consisting of tens of thousands of nodes, each with individual properties and connected through heterogeneous networks, an efficient mechanism is required to access, provide and control information flows in the ecosystem. To cope with the diversity inherent in the ecosystem, the user or the application programmer should be able to focus on the key attributes that uniquely determine the characteristics of the modelled system. In other words, the user should be able to state what he wants to accomplish without being specific how the underlying system will execute it. A flexible interface is required to support multiple applications with differing requirements. The interface must support the achievement of the following goal:

> *Goal:* Given an interest I in data and a set of quality criteria C, find a formal representation of interest and quality criteria. Allow the injection of I and C to retrieve corresponding information from the system. Ensure that requirements of multiple applications are supported. Provide the flexibility to gradually adjust the requirement during the lifetime of the system.

During the scenario analysis at the beginning of this chapter a language interface was suggested as tactics to achieve the identified quality attributes. Languages are the most universal interfaces available. In the context of technical systems, coupling between components or services can be achieved by providing a language with shared vocabulary. The completeness of a language, i.e. the ability to principally express any computable function in a language, can be formally proven by implementing a Turing machine in the language. Hence practicality and suitability of a language can be evaluated qualitatively as well as quantitatively.

Declarative programming languages allow to specify the logic of a computation without describing its control flow. Hence they fulfil the requirement to specify the "what" without the "how". Widely known examples of declarative programming languages include Prolog, Haskell and Linda. The Structured Query Language (SQL) [67] is partially, e.g. SELECT FROM TABLE .. queries, a declarative language. Although SQL became a de facto standard considerable issues have been identified, e.g. in [30]. However, fact is that SQL is easy to understand, simple to integrate and has a large developer community. This observation is also supported by the emergence of new, on the SQL based, query languages such as Google QL (GQL) [55] and Salesforce Object Query Language (SOQL) [45]. Taking this into consideration, in the following, the Service eCoSystem Query Language (SCSQL) based on the SQL is introduced as interface for the ecosystem. The brief introduction focusses on the core concepts. The language is designed such that a developer

with basic knowledge of the SQL should be able to start immediately. However, the full version of Extended Backus-Naur Form (EBNF) of the grammar is provided in Appendix B.

6.5.1 Foundations

The language is data centric, therefore most types, operators and statements are designed to store, update, retrieve or route data from one or more sources to one or more specified sinks. The query execution and optimisation subsystem as explained in Section 6.3 allows for location transparent management of data. This means that the programmer does not need to know where or how the data is stored. He neither needs to know if it is stored on one or more nodes nor whether it needs to be delivered to a single or multiple nodes.

Listing 6.2: Declarative routing from source nodes to receivers

```
S-> SELECT * from NODES
    WHERE    location = 'LOC_A'

R-> SELECT * from NODES
    WHERE    type = 'HMI' AND locations = 'LOC_B'

M-> SELECT voltage FROM S
    WHERE  voltage > 225V
    WINDOW(0,1h,10s)
    RECEIVER R
```

The example Program 6.2 demonstrates this capability of routing information declaratively. The first statement selects a set of nodes located at the specified location. The `->` operator binds the set to a variable `S`. The binding operator has a similar semantic as a pointer in the *C* programming language. Data selected by the select statement is not copied to a local memory, but rather `S` points to the data which may be located at another node or even distributed to more than one node. Transparent to the programmer, the underlying runtime system ensures the availability of data at the right time, place and quality. The second statement selects a set of receivers to fulfil certain criteria. In this particular case, they must be of type Human Machine Interface (HMI) and located at a specific location. Eventually, the third statement opens a measurement stream from the source nodes to the specified sinks. Stream elements are filtered to readings above the specified threshold. The execution semantics of Program 6.2 is sequential, i.e., `S R M`. However, optimisations may yield a different but semantically correct execution order which remains transparent to the programmer (s. Section 6.3).

In certain situations it is convenient to explicitly address a node or set of nodes. Therefor, SCSQL provides the `@`-operator. Anything to the left of the operator is interpreted as

variable, or table, while anything to the right of the operator is either a variable or a table specifying one or multiple nodes[3]. During optimisation, queries bound with the `@`-operator are excluded from the execution candidate node set and are fixed similarly to sources (s. Section 6.3).

The `window` parameter of the `M` statement has also implications on the execution of `R` and `M`. Since `M` depends on the availability of `R` and `S`, `R` and `S` may be executed multiple times such that the timely availability of information is assured. However, `R` and `S` are only executed during the window of activity.

6.5.2 Programs

The program concept groups queries and declarations into reusable modules. Programs may consist of multiple declarations, queries and user defined aggregation functions. A program is defined using the `Program` keyword followed by a unique *name* or *identifier*. The name is used to manage the program, i.e. control the execution life cycle, retrieve results or remove it from memory. Program statements are enclosed in curly brackets as shown in example listing 6.3.

Listing 6.3: Example program

```
Program helloWorld {

    S -> SELECT *
         FROM   NODES
         WHERE  type = HMI

    UPDATE S
    SET    display = "Hello World!"
}
```

Besides providing the popular “Hello, World!” example, Listing 6.3 underlines once more the data centric character of the language. SCSQL does not have specific input/output functionality. Instead it focusses entirely on organising, storing, updating and moving data. Hence, in Listing 6.3, selected nodes of type HMI are passed a piece of data, in this case a string, to set a variable referenced by “display”. Receiving this information, the nodes decide how to handle it, e.g. choosing a method to display information on a screen, logging it to a database or simply discarding it.

[3]Variables or tables must at least include the node ID.

6.5.3 Queries

Queries are specified in SQL like semantics. Standard select statements can be extended with `WINDOW(from, to, interval)` and `RECEIVER <NODESSET>` statements. The window statement specifies the activity window of a query. The query is activated when the time specified by `from` is reached. Execution stops on `to`. During execution new values are emitted every `interval`. Logically it follows $to < from$. If $\frac{from-to}{2} < interval$ the query is executed exactly once. If $interval == 0$ a new value v is emitted for every $v_{t+1} \neq v_t$ whereby the difference is measured and detected by the host node. The detection is non deterministic and no temporal guarantees are assured.

The `RECEIVER` statement takes either a single node or a set of nodes as parameter. Data delivery is planned and optimised by the execution subsystem and therefore transparent to the programmer. If the node that received the program is not a member of the receiver node set, it most likely will not receive any of the specified measurements as they are forwarded directly from source to sink.

6.5.4 Numbers

SCSQL supports integers and floating-point numbers. The integer type, INT, is signed and exact, while the floating point type, Double, is signed but approximate. Integer values range from -2^{31} to 2^{31}, Doubles are 64-bit double precision in correspondence with IEEE 754 [69]. Integers and doubles are declared using the `INT` *and* `Double` keywords.

Listing 6.4: Numbers

```
INT abc = 1
Double def = 2.0
```

6.5.5 Strings

Unlike in other languages, strings in SCSQL are not sequences of a primitive character type but the String type is primitive itself. Strings are declared by the keyword `String`. A string can be concatenated by the `+` sign (Listing 6.5).

Listing 6.5: Strings

```
String abc = "ABC"

abc = abc + "DEF"
//abc = ABCDEF
```

6.5.6 Conditional Execution

SCSQL supports common control structures found in most programming languages including `if,else,then` and `if,else,then,if` constructs. A `FOREACH` loop construct allows for iterative computation on result sets.

All standard conditional boolean operators, i.e. `AND, OR, NOT` are supported. In addition to boolean operators, comparison with `==,<=,<,=>,>`, i.e. equal, less than or equal, less than, greater than or equal and greater than, methods are supported.

The use of boolean operators and comparison methods is limited to the WHERE section of a SELECT statement and the body of user defined functions is explained in the following section. Note: the runtime system does not enforce real-time behaviour if programs include non-linear or conditional statements.

6.5.7 User Defined Funtions

Additional to the standard aggregation functions MIN, MAX, SUM and AVG, users can define their own functions. Listing 6.6 provides an example of a simple user defined function which filters all values outside a band defined by two thresholds.

Listing 6.6: Strings

```
FUNCTION Filter(INT val) {
    INT high = 10
    INT low = 3

    if (val > high OR val < low) {
        RETURN NULL
    }

    RETURN val
}
```

User defined functions can be used as regular aggregation functions. Values are passed to the function as they occur in the selection. The function is called for each row matching the query.

6.5.8 Quality Attributes

Queries can be annotated with hints on the quality that is expected from the produced result set or stream. The keywords for quality attributes are: `validity, accuracy, encrypt, sign` and `restrict`. Each of the keywords implements

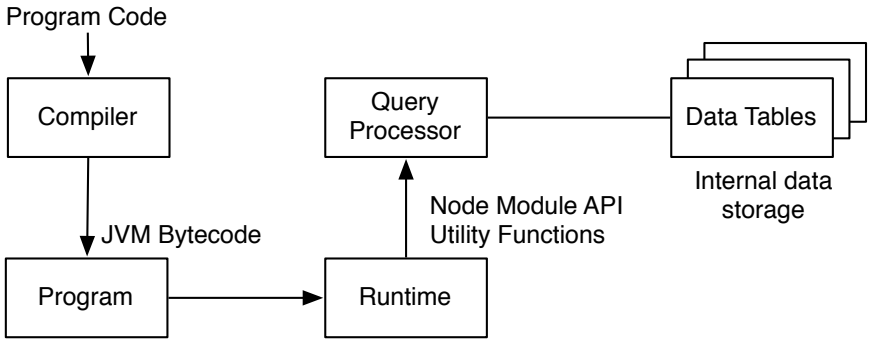


Figure 6.19: Compiler framework architecture

the quality attribute to the corresponding description in Chapter 5. *Validity* takes as argument an integer which determines the minimum time in milliseconds an item must be valid on execution. *Accuracy* takes either `HIGH` or `LOW` as argument and controls whether hard- or soft real-time processing is enabled. The security attributes cause message content to be encrypted and signed respectively. The `restrict` keyword limits access to certain roles.

6.5.9 Compiler Architecture

SCSQL programs are compiled using a flexible compiler framework. Programs are translated into an internal, language independent representation which can be translated to multiple target systems. The compiler starts by parsing user defined program source code and generates an Annotated Syntax Tree (AST). From the AST, target code, e.g. Java Virtual Machine (JVM) byte code, is generated. The process is illustrated in Figure 6.19.

The generated byte code contains all objects and methods necessary to execute the query on a target node. The runtime block adds additional functionality for standalone execution. Hence, programs are converted entirely to run on a target system. Depending on the target, the program may include the entire runtime environment[4] hence requiring no additional software.

The advantage of generating JVM byte code is that it is relatively compact, i.e. a single *.class* file contains the entire program including all failure handling, communication etc. Additionally Java is supported by a large variety of platforms ranging from servers to handheld devices. Although possible, generation of byte code other than Java would most likely be less compact as additional functions like type conversions and marshalling code would have to be included.

[4]For Java it might make sense to assume that at least a JRE is in place

Once generated, programs are either packaged and transferred, in case of standalone deployment, or, if handled by a node, scheduled and sent by the communication module to the target node(s). In both cases query execution is handled as outlined in Section 6.3.

6.6 Implementation View

While previous sections described the data-centric architecture for large-scale industrial systems, this section illustrates selected components of an example implementation. The purpose of the discussion is to provide an implementational view on key aspects and interfaces to address technical issues. Provided information may serve as a guide for future extensions and addition of modules.

6.6.1 Asynchronous Request Handling

Node functions operate asynchronously. Triggered by the event kernel, a module retrieves state from the memory module, computes the next step and stores the state back to the memory. Surely, reading and writing to the memory is implemented by reference such that no data is actually copied.

The event-based architecture allows the integration of additional modules. However, custom modules must obey conventions to not undermine the processing concepts of the node.

- A new module must not block in any way. No loops or waits are allowed.
- Modules are strictly forbidden to use threads or wait for a lock to be released.
- Except the memory module, modules are not allowed to store state. Upon activation modules retrieve state from memory, process and store the state back to memory.

The processing model of a node is single threaded with the exception of the communication module which maintains its own thread pool with client and worker thread for asynchronous I/O. Low level networking is handled by the Netty (www.jboss.org/netty) library which provides an asynchronous event-driven network application framework. The communication module queues incoming and outgoing messages, thereby encapsulating and de-coupling the I/O and kernel process threads.

The communication module is one of the modules most likely subject to extension. Adding new overlays or security layers requires knowledge of the internal implementation. Figure

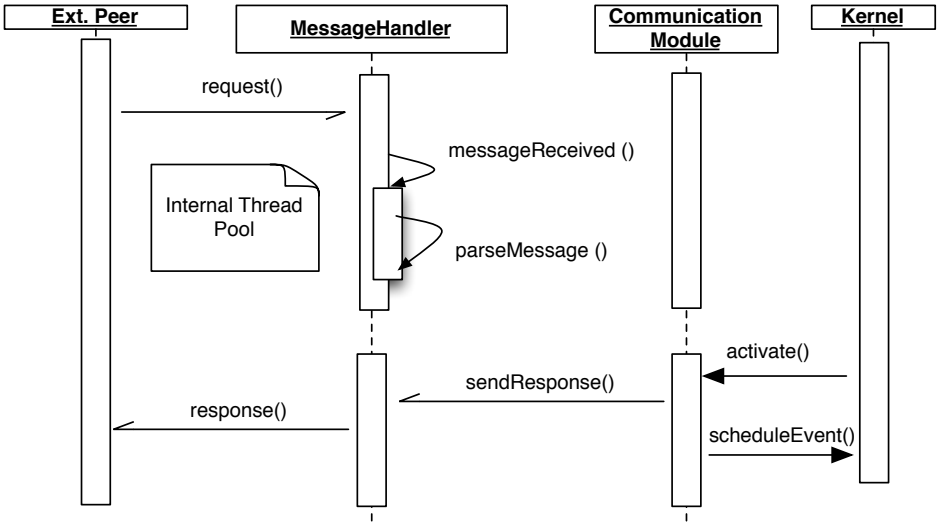


Figure 6.20: Asynchronous message handling

6.20 illustrates, in greatly simplified manner[5], the message handling process. Requests arriving at the node's server socket are dispatched, parsed and queued into an internal message queue. This is handled by the internal thread pool. Under high load situations the communication module might decide to discard incoming request messages. Response messages, however, are never discarded. One could argue that this behaviour compromises the real-time capabilities, yet no system with fixed capacity resources can guarantee real-time behaviour in the face of overload. However, overloading is unlikely as it would mean that, e.g. during a denial of service attack, simultaneously large numbers of query requests arrive at the the node. In any case, real-time nodes should be engineered such that they are shielded in a security domain and hence cannot overload.

Parsers and message decoders are part of an overlay component. Messages are mapped according to their type header. However, no further action is taken by the overlay until the activation by the kernel. During activation, overlay modules examine the content of the message and decide on further actions. Due to the restriction to avoid looping and blocking during kernel activation, message examination instruction is rather simple. It basically checks the type of the message and generates events to further process it. If the message cannot be mapped to an overlay it is wrapped into an event and passed to a respective module, i.e. either custom or query processor (type 0). The internal thread pool of the communication module also dispatches the outgoing messages queue from the internal out queue. The outgoing queue is filled either by the kernel, e.g. responses, result

[5]The message handler is actually split to several classes, i.e. RequestMessageHandler, ResponseMessageHandler, MessageParser, MessageEncoder, MessageDecoder and EventGenerator plus utility functions.

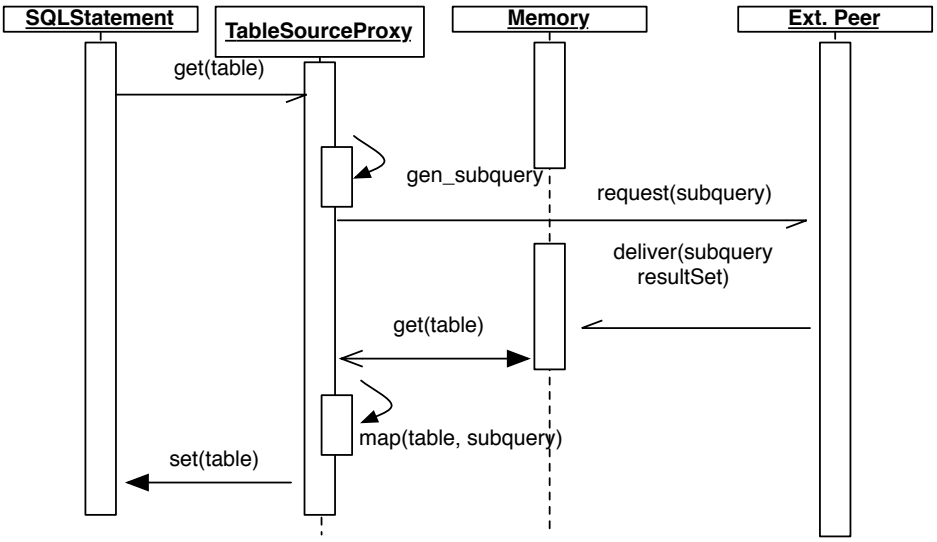


Figure 6.21: TableSourceProxy

sets, monitoring or internally triggered by failures to parse messages or unknown types.

6.6.2 Data Discovery

In the easiest of cases, all data necessary to execute a query is already contained in the local memory. If, however, the query references a table residing in the memory of another node, or the data contained in memory does not meet the quality requirements, the query needs to be extended by one or several sub-queries to fetch missing data.

Sub-query creation is initiated by the query optimiser if data is not available at the time of scheduling. Queries are compiled to an SQLStatement object which in turn accesses (events) the memory component via a TableSourceProxy which will detect the missing data using `readData` instructions. In Figure 6.21 the process is illustrated. Upon execution the SQLStatement requests table data from the TableSourceProxy. If the table is not in memory, the proxy triggers the optimiser to reconfigure the queries. The proxy maps the result set of the sub-query to the table requested by the local SQLStatement, hence, the remote access remains transparent to the SQLStatement. Listing 6.7 further illustrates the case by providing a concrete query and the corresponding sub-query.

The first statement is the query as injected. As table2 becomes unavailable, a sub-query is generated which retrieves the required data, in case of the example v1 and v2 to local memory. As optimisation the WHERE clause could be transferred for execution on node2. In this case only v1 would be transmitted and the evaluation of the main query would be rewritten to evaluate the WHERE clause only on table1. For the clarity of the example

Listing 6.7: Example query targeting multiple data sources

```
//main query
S -> SELECT     v1
     FROM       TABLE table1, table2
     WHERE      v2 < 10
     WINDOW     (0,100,10)

//generated subquery
TABLE2 -> SELECT    v1, v2
          FROM      TABLE table2@node2
          WINDOW    (0,100,10)
```

table names are chosen to be short and descriptive. In a productive setting, tables would be named using namespaces to avoid conflicts.

6.6.3 Role-based Access

Depending on the type of devices that host a node, different security mechanisms may be appropriate. In certain cases an authentication mechanism executed by the communication module may be enough to prevent unauthorised use of the node. In other cases more fine-grained access may be desirable, e.g. automation equipment at a tie line. The TSO owning the device requires full access to the functionality. However, in order to support fully automated business processes the owner may want to allow other TSOs to directly receive data from a field level device. Write access, however, shall remain forbidden. In this section, a fined-grained Role Based Access Control (RBAC) mechanism for node data is introduced. The concept is based on Shibboleth (http://shibboleth.internet2.edu), an attributed-based authorisation service emerged from the Internet2 community. One of the initial goals of the Shibboleth project was to support cross-organisational identity federation. Hence, it is well suited for cross-TSO device access. RBAC solutions like Shibboleth are rather complex and, thus, are not covered in detail by this work. Extensive material is available in related publications and the community web site (http://shibboleth.internet2.edu).

Figure 6.22 shows a simplified version of the authentication process. For the scenario, we assume a query that is issued at a node *inside* the organisation the node is located but requesting information that is located at an *external node* which hosted by *another* organisation and protected by the security mechanism.

As a prerequisite, stakeholders need to exchange security certificates (e.g. X.509) and negotiate role attributes, i.e. define which roles exist and how they are named. Once the attributes are fixed, no further manual adjustments regarding the user/role mapping are required. Instead each stakeholder manages role memberships locally by using already

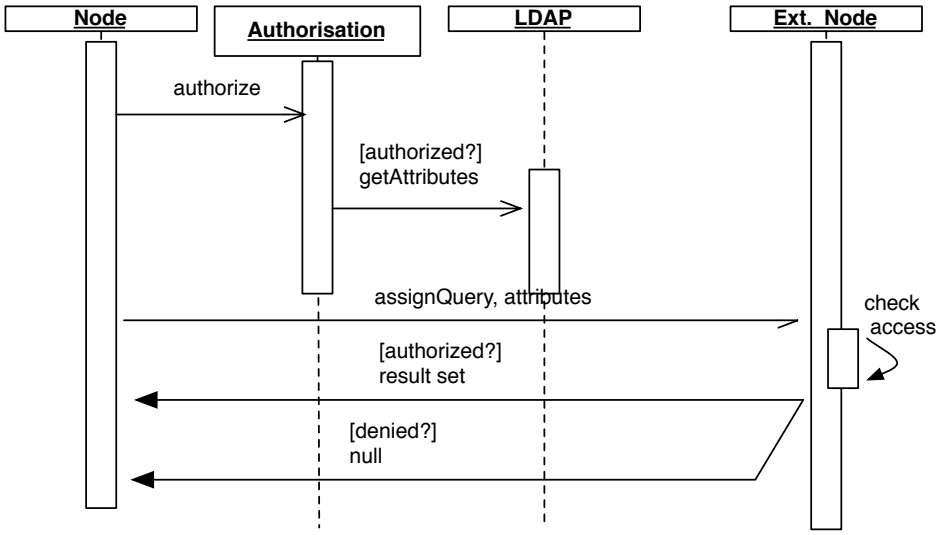


Figure 6.22: Role based access

existing infrastructures and services, e.g. LDAP, to assign role attributes to users. Consequently, user requests are authenticated locally at each stakeholer. The authorisation process executes as follows:

1. A query is issued to retrieve data from a secured node outside the organisation
2. The query issuer authenticates at the local authentication service
3. The query plus the signed role attributes are sent to the remote node
4. The node verifies the signature and checks whether the associated role is allowed to access the requested data
5. Depending on the previous step, the query is scheduled locally or the request is denied

Authorisation is done for each request; hence the procedure is slightly different from the standard Shibboleth process. Authorisation attributes are encoded in a message. Communication modules supporting the authorisation method validate the attributes and pass them to the memory component in form of a program. The initial program sequence performs the security checks. If access is denied, the program finishes and creates an empty result set. Otherwise the query is executed regularly. If the query is continuous, at the beginning of each iteration, the security sequence is executed. A continuous query aborts if the security rules were changed and the check failed. The failure is reported back to the caller.

6.7 Summary

This chapter presented a data-centric architecture for large-scale industrial systems. Based on two scenarios, detailed requirements and tactics to satisfy these requirements were deduced. Using the tactics, the architecture consisting of loosely coupled distributed query processors has been developed. The three key concepts of the architecture are (i) the modular node built of five essential modules, (ii) the index cloud providing global access to nodes and data, and (iii) SCSQL, the data-centric query language to query, structure and route information between architecture components. The tactics *maintain semantic coherence*, *generalise the module*, *information hiding* led to the five module architecture. The query processor implements *manage event rate*, reduce computational overhead, *fixed priority scheduling* by its optimisation capabilites. The event kernel results from the tactics: *introduce concurrency*, *fixed priority scheduling*, *virtualise the processing kernel*. The communication module corresponds to *virtualise the network* and security aspects, i.e. *data integrity*, *authorise users*, and *confidentiality*. The concept of the index cloud implements *use an intermediary* and *runtime registration*.

The event-based processing model does not only ensure real-time capabilities but also allows for extensions by further custom modules. At the heart of a node operates the query processor which configures execution of queries and manages multiple query life cycles. Node capabilities can be extended by adding custom modules and device drivers for specific hardware functions like sensors and actors. Concentrating all state in a single module, allows the query optimiser to include information on other queries, state of communication, resource consumption, entropy and current complexity. The benefits of this approach will be further examined in the following chapter.

Instead of propagating an entirely de-centralised approach, two general node classes are differentiated: index nodes and regular nodes. Index nodes are determined to maintain a global meta data index of all nodes currently participating in the ecosystem. As the availability and consistency of meta data information may be critical, the index cloud features several write modes from eventual consistency to atomic writes. The index cloud is able to scale with the number of nodes participating in the ecosystem. It automatically configures itself to read and write zones in order to serve varying read and write loads.

Designed to scale to thousands of nodes, the architecture requires efficient means to access information as well as monitor the health of the system. The high level query language, SCSQL, provides a pragmatic language based interface to declaratively specify interest in information. The compiler framework is able to generate target byte code and enables developers to write application specific code at a high level of abstraction and without the need for further configuration or deployment overhead.

The following chapter will evaluate the presented architecture by the quality attributes identified in Section 6.1. It implements the architecture evaluation methods described in Chapter 4. Additionally, the performance of the index cloud is investigated and the stability of the processor network is discussed. Although the benefits of the query language are rather qualitative, e.g. encapsulation of complexity, its impact is evaluated by proving its capability to express arbitrary functions.

Chapter 7

Evaluation

In contrast to large-scale systems in other domains, e.g. media or communication, the industrial domain has particular (non-functional) requirements, e.g. stability and safety, that must be addressed essentially. The concepts and architecture developed in Chapters 5 and 6 are designed to meet these requirements.

Applying the evaluation methods presented in Chapter 4, this chapter evaluates the applicability and suitability of the architecture. Subsequent to the evaluation of the architecture itself, selected components are investigated with regard to their complexity, performance, availability, reliability and stability. Finally, the expressiveness of the programming language is proven. Applying the language to a standard problem, i.e. a control loop, its suitability, compactness and implications to modifiability are shown. The chapter concludes with a summary of key findings.

7.1 Architecture Evaluation

Implementing the methods introduced in Chapter 4, this section evaluates the architecture presented in Chapter 6. The aim of this evaluation is to (i) demonstrate the suitability of the architecture for the systems under investigation and (ii) to elaborate the superiority of the design chosen versus alternative approaches. The evaluation is based on the scenarios taken from Section 6.1 and the quality attributes summarised in Tables 6.1 and 6.2.

7.1.1 Identification of Architectural Styles

Architectural styles, also called architectural approaches, are commonly used [11] to describe the architectural aspects of software quality. In this first step of architecture evalu-

ation, the architectural styles of the architecture presented in Chapter 6 are identified. In correspondance to the methods presented in Chapter 4 a qualitative analysis is conducted to elaborate wether architectural decisions support functional as well as non functional requirements.

1. The architecture is *event-based*. The node kernel manages the inter-module communication by scheduling and dispatching events. Events can be both external, e.g. new measurements, or internal, e.g. scheduled maintenance events. Using memory tables, events can be broadcasted to one or more components. At global scale, nodes register subscriptions for particular events. They are informed if matching data items are added, changed or purged.

2. The architecture is *data-centered*. Local and global nodes tables are used as *blackboards* to exchange information between nodes and individual queries and programs. Other tables and references to tables on other nodes can also be used as blackboards. The state of a node is comprised of the tables contained in memory. Considering the replication capability of the index node as well as the support for various consistency models, it may also be interpreted as *repository* [11].

3. The query processors manifest a *virtual machine style*. The query language introduced in Chapter 6 is Turing complete (s. Section 7.5.1). Hence, arbitrary functions can be implemented on top of the query processor.

4. The architecture is *layered*. On the lowest level of abstraction, the communication module abstracts from the specifics of the physical network and the hardware drivers encapsulate particular hardware characteristics. The middle layer is comprised of monitoring, kernel, query processor, and memory. At the top layer resides the application logic, i.e. the queries or programs developed in the query language.

5. The architecture uses an Aggregator-Escalator-Peer [60] style. Nodes contributing to the execution of a query are constantly monitored by the node managing the execution. It aggregates the states as well as local measurement to re-optimise the execution.

6. The architecture follows a *pipes and filters style*. Since, in fully replicated mode, index nodes do not share state, functionality between query nodes and index nodes can be added transparently. Hence, additional caching, authorisation, partitioning or compression features can be added on demand.

Table 7.1: Evaluation of architectural styles for quality attributes

Style	Performance	Predictability	Modifiability	Security	Scalability	Flexibility	Stability	Observability	Awareness	Availability	Integrability	Testability	
Event-Based	+	+	+		-	+					+	+	
Data Centred			+		+			+		+	+		
Virtual Machine			+	+									
Layered			+	+								+	
Aggregator-Escalator-Peer							+	+	+				
Pipes and Filters	+		+	+	+	+	+						

7.1.2 Influence on Quality Attributes

Recalling the scenario descriptions in Chapter 6, the following quality attributes were identified: performance, predictability, modifiability, security, scalability, flexibility, stability, observability, awareness, availability, integrability and testability. Using the architectural styles identified in the previous section, the contribution of each style to the achievement of the quality attributes is now analysed.

Table 7.1 summarises the evaluation. A plus (+) sign indicates positive influence, a minus sign (-) negative influence and a blank indicates that the style has no influence on the attribute.

For this evaluation step it is distinguished between system and node level. The latter focussing on the software architecture of individual node, i.e. the modules and their interactions as described in Section 6.3. The former treating index nodes and query processors as holistic architectural artefacts, i.e index cloud and distributed query processor.

Event System

On the node level, the event system style contributes to performance as it limits the amounts of threads to be handled at the server. Using asynchronous handling of client requests allows to serve a high number of concurrent clients. Additionally, by the discretisation during event scheduling, the outcome of the execution of event code becomes predictable. The event system style decouples the node modules which communicate solely asynchronously and through an mediator, i.e. by scheduling events with the kernel. Hence,

modules can be exchanged, divided or aggregated without adjustments on the sender side. On the system level, the event system style enables integration. External applications can register subscriptions at the index cloud to be informed of system modifications. Since subscriptions declaratively describe an interest in data modifications, individual sender and receiver components are completely decoupled. Having each execution step as discrete object together with a time of execution allows to record event streams for later replay in controlled environments. Hence, the approach contributes to testability.

Data Centred

On the system level, the data centred style (blackboard) contributes to integrability by providing a unified and standardised interface to heterogeneous and highly distributed data. At the node level, the memory constitutes the local blackboard for local queries to exchange information. Since individual data stores can be accessed by other nodes, they act as caches and enable replication. Hence, availability and scalability is increased. Furthermore, the local memory stores information on the state of the node in the form of tables. Thus observability is supported by the unified access to state information in the memory.

Virtual Machine

The virtual machine enables the portable execution of programs. Since the execution environment can be fully controlled, i.e. by the limited instruction set, the style contributes to security of individual node and the entire system.

Layered

At the node level the layered style organises modules such that in each layer a higher level of abstraction is achieved. The communication module and device drivers abstract from the physical network and underlying hardware allowing other modules to address data in a transparent and declarative manner. Kernel, memory and query processor constitute the basis of an execution system which can be programmed with a high level programming language. Programs written in this language operate at the highest level of abstraction. Layers are separated by well defined interfaces and hide concrete implementations. Consequently, individual layers can be exchanged leaving the modules in other layers untouched. This not only contributes to modifiability but also to testability, as, e.g. individual modules can be replaced by an event recorder/player or the communication module could be replaced by a network simulator.

Aggregator-Escalator-Peer

Using the aggregator-escalator-peer style, the autonomous, self-organising, self-healing and self-optimising behaviour of the system is supported. The monitoring component retrieves information of all nodes currently participating in a controlled action. Based on this aggregated information, nodes can re-optimise the current execution. It hence supports for stability, observability and awareness.

7.1.3 Quality Attributes

While the previous section showed how the architectural styles support identified quality attributes, in the following, contributions to the achievement of the quality attributes by respective architecture components is discussed.

Performance

On a qualitative level performance is achieved by the event-based processing model. Depending on the underlying operating system, thread management can take considerable processing time. In contrast, the single threaded processing model of a node uses a minimum of resources. The ability of the SCSQL framework to compile queries into standalone programs that run natively on a target system circumvents complex middlewares and hence contributes to performance quality. On a quantitative level Section 7.4.1 elaborates processing performance under varying workloads.

Predictability

The kernel forces modules to schedule computation on fixed, discrete time events. Prior to scheduling queries and programs, the kernel checks whether resources are available and time limits can be met. Hence, once a program is accepted by the kernel, it is guaranteed to meet the defined limits. If the kernel, however, is implemented in user space it is dependent on the predictability attributes of the underlying operating system functions. If deployed on automation equipment, which often is operated by real-time operating systems, the kernel is implemented as single real-time task and hence inherits the real-time behaviour of the OS.

Blocking I/O as in the communication module is encapsulated and, in combination with the strict scheduling of the kernel, cannot affect stability of a node. In context of overload, e.g. during Distributed Denial of Service attacks (DDoS) predictability and real-time

qualities will be affected. This problem is not unique to the architecture but applies to all real-time systems. If the capacity of resources is reached the service cannot be maintained. A node aims to shield itself from overload by denying requests that conflict with schedules. If, however, message queues in the communication module overflow important messages, e.g. related to query reconfigurations, might get lost. Like in any industrial system, it is the responsibility of engineering to shield critical components from overload such that the safety of the installation is not compromised.

Modifiability

The node architecture constitutes a minimal yet universal compute model. Kernel and processing system can be implemented with low implementation complexity[1]. Hence, the effort required to stabilise the code base such that it is able to operate stable for decades is manageable. The universality of the language interface allows for device updates at a high level of abstraction and without the danger to compromising the stability of the node. Hence, nodes can be updated to run application specific code. This supports the implementation of evolving requirements and changed operation patterns. The universality of the language is formally proven in Section 7.5.1.

Security

Security has several dimensions. First, opening nodes to run arbitrary programs developed in SCSQL might allow attackers to inject malicious operations that prevent the node from fulfilling its application tasks. Second, data transmitted between nodes can be intercepted, manipulated or discarded. Third, flooding a node with requests (denial of service attack) may hinder the node to service regular requests.

The primary interface of a node is the communication module. Internally it is implemented as layered architecture to abstract from lower level networking towards message and event-based processing. This allows for the implementation of standard authentication protocols, e.g. login/password based access or private/public keys as demonstrated in Section 6.6.3. In the same manner protocols like Secure Socket Layer (SSL) can be used to encrypt and sign the traffic between nodes. Which method is chosen for a concrete deployment, depends on the hardware and environment the node is operating in. It might be greatly differing from a substation scenario where SSL might be impractical due to resource limitation and a control centre where the node operates on a server with dedicated hardware to process SSL communications.

[1] In the prototype implementation less than 3500 lines of code

The proposed node architecture is, from the SCSQL developer's point of view, a closed environment which does not allow to access lower level operating system functions other than defined by respective (custom) module interfaces. If the target architecture is a Java virtual machine, additional security measurements apply.

Adaptability

Adaptability has been identified as quality attribute in the context of scalability, flexibility and stability. With the concept of networked query processors, individual queries can be broken into sub-queries and distributed to remotely available resources. Hence, as a core feature, nodes scale with the number of available resources. The index cloud is capable to adjust its structure in order to reflect different load characteristics. This feature is also quantitatively evaluated in Section 7.4.

The parameter driven query optimiser contributes to the achievement of flexibility qualities. Having the entire state of the node concentrated at the memory module, allows the optimiser to assess the current state as well as the state evolution, i.e. state changes over time. It can further reconfigure the query load in order to reflect current environmental conditions.

Stability has been partially covered by the previous discussion of modifiability and security. The ability of the kernel to shield the node from uncontrolled resource consumption ensures that programs are executed complying with all specified quality constraints. Restrictions apply in the context of overload as discussed previously. Processing stability is further discussed in Section 7.3.2.

Observability and Awareness

By the language based interface and the concentration of state in the same data model as other data, nodes provide a unified interface to their current condition. The query optimiser uses this information to decide whether the migration of a (sub-)query to another node will yield more efficient resource utilisation or destabilise query execution.

Apart from its own state the node is guaranteed to have up to date information of the current state of its neighbours. The monitoring component gathers state relevant information from other nodes and detects node failures. Based on the collected data, the optimiser can create detailed statistics in order to determine the best execution candidate in terms of reliability and reputation.

Availability

The minimalist processing environment as well as the kernel functions to protect core programs ensure the stabile operation of all nodes. The inherent ability of the optimiser to transparently relocate query execution ensures that, even in the event of individual node failure, query execution can continue. The index cloud provides a watchdog mechanism to detect master failures and seamless failover. Data stored in the index cloud is replicated such that it remains available even if individual nodes fail. Node availability is further investigated in Section 7.4.3.

Integrability

Large-scale systems are rarely built from scratch but rather emerge from the integration of existing systems. SQL is the query language used in almost any[2] business application. The query language provided in Section 6.5 is based on SQL and hence supports the major constructs of the language. The adaption to industry standard connectors like JDBC is straight forward. Therefore vertical integration from the field level, i.e. the nodes, directly into the business process is possible. From the perspective of the business application the entire network of nodes including the index is represented as a single database with tables and data objects similar to a regular RDBMS. Using standard SQL queries, the application is not limited to data access and manipulation but may also describe data flow between nodes and other infrastructure, e.g. warehouses and archives. Using the extensions of SCSQL, applications are additionally enabled to enforce quality constraints, conveniently express information routes and pre-process large volume data streams in-network.

Testability

By the event-based kernel, node modules are loosely coupled. This allows to replace a module seamlessly with test modules or record event streams for later debugging in a controlled environment. Hence, entire systems with thousands or millions of nodes can be tested before their deployment. Therefor, the simulation tool described in Chapter 4 is an ideal candidate. To simulate a node, kernel and communication module must be replaced with their simulation counterparts. Since the processing model avoids extensive thread usage, thousands of nodes can be simulated on commodity hardware. The discrete event processing of the kernel allows for deterministic and reproducible simulations.

[2]Every application that uses RDBMS as database

Ecosystem Concept

Since the scenarios are based on use cases of today, the openness and de-central organisation of the ecosystem were not identified as quality attribute. However, in short to mid term the need for alternative engineering methods for large-scale systems will become immanent (compare ULS section in Chapter 2). Hence, this section evaluates the openness and data model of the ecosystem. Moreover, it is shown how the architecture implements the three core services identified in Chapter 5.

The focus of this work, and a key driver for the ecosystem concept, is the achievement of timely delivery of data. The sole information that a certain sensor exists, that it has a particular hardware address, that it is manufactured by a specific vendor is not relevant for most operational processes. What matters is the sensed value upon which control decisions are made. Together with the raw data, meta data describing type and accuracy might also be useful. The ecosystem for energy services is aimed to meet these requirements for future large-scale systems. It abstracts from low level information aggregation and implements data access at specified qualities of service.

The data and table model allow to structure data yet leave flexibility for data evolution (changes to structure and quality requirements over time). For example, if the meta data of a device is published to the index cloud, it is stored in the nodes table using a column for each attribute of the meta data. If, at a later point in time, device meta data changes, e.g., because, the device was upgraded with a new feature or a feature previously not included in the description is added, the description is updated in the data cloud without the need for a schema update of the entire data base.

The architecture of Chapter 6 supports the three core services (identification, registration and incentive) required for successful operation of an ecosystem. Identification is implemented by the communication module. Sender ID as well as conversation ID map conversations to concrete ecosystem entities. Optional security mechanisms such as the one provided in Section 6.6.3 further extend this features by identity management functions. The index cloud acts as global registry for all data available in the ecosystem. It allows to publish and search for nodes, services or other general data. The architecture does not implement explicit services for financial transactions. However, the security mechanisms of the communication module, the storage functionality of the memory module together with the discrete processing style and the SQL-based interfaces enable seamless integration with standard accounting and transaction systems. Negotiations, trades and transactions are logged internally, e.g. in transaction tables. Integration similar to the vertical integration scenario of Chapter 6 allow further processing in standard business process engines.

In Section 5.6.2 the requirement of having the right data at the right place and time was concretised by specifying a set of quality attributes for ecosystem data. SCSQL supports the implementation of the attributes by providing means to formally represent them as query statements. Moreover, the corresponding runtime system enforces their achievement (s. Section 6.3).

7.1.4 Summary

Previous sections evaluated the architecture presented in Chapter 6. Following the classification and contribution of individual architectural styles, each of the identified quality attributes where discussed and their achievement confirmed. Also the limitations of the architecture in the case of overload were illustrated.

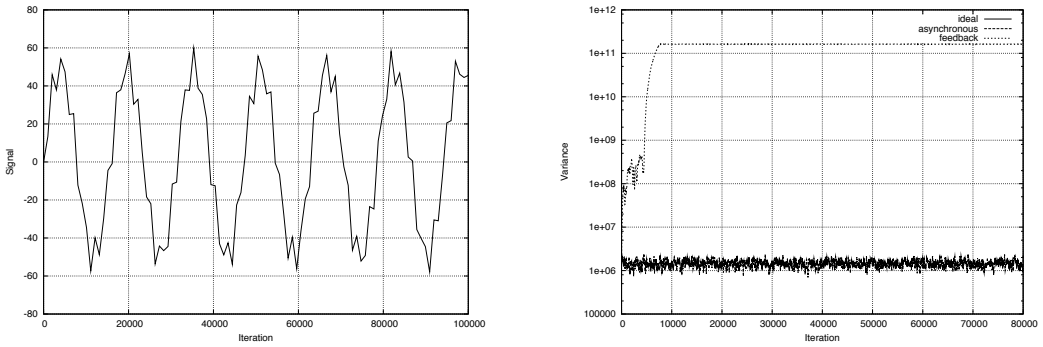
In the following, key components are evaluated quantitatively. Therefor, the simulation methodology described in Chapter 4 is applied. For each experiment, environment and resource models are explained. However, before the discussion starts, a metric for a quantitative measurement of complexity is introduced in the following section.

7.2 A Measure of Complexity

For the quantitive evaluation of the architecture a simple complexity measure is introduced. The measure is based on the observation that complexity in a distributed system is determined by the degree of distribution, i.e. the number of elements contained in the system, the diversity of the elements and the dependencies between the elements. This can be shown with a simple experiment. Consider a distributed system[3] with nodes connected by some kind of network. An external event occurs that affects all nodes simultaneously[4]. The challenge for the nodes is to log the event as measured. However, they do not log the event locally but at another node, hence they have a simple dependency. In an ideal world event logs at each node would contain exactly the same signal after the experiment. Considering the degree of distribution, diversity and dependency, the ideal world assumption is not realistic and one would expect variations in the records. To provide a metric to measure the degree of variations, the variance of measurement inherent in the system is used. Each measurement reading corresponds to a state of a node. The variance of states, therefore, provides a metric on the number of different states which, in this work,

[3] A network of query processors.

[4] Focussing on the complexity of the system it is assumed that the event reaches the nodes exactly at the same time. This might not be possible in nature.



(a) Signal (b) Variance

Figure 7.1: Signal and variances in an example system

is defined as complexity of the system.

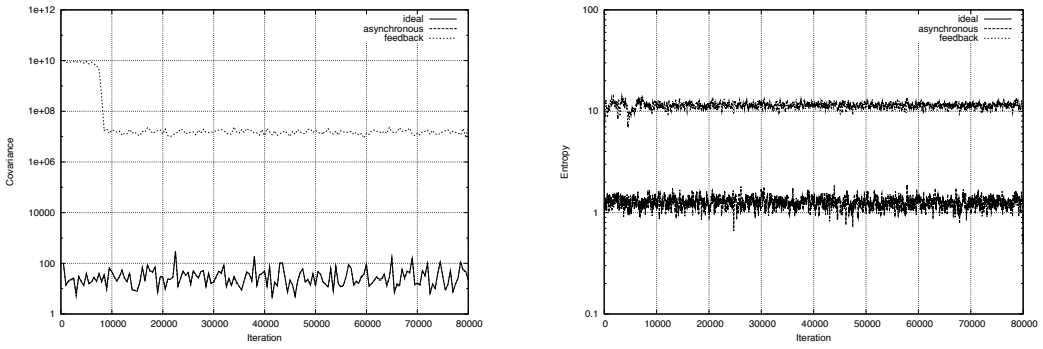
Figure 7.1 illustrates an experiment with ten nodes. The signal as shown in Figure 7.1a is applied to the system. Figure 7.1b shows the resulting variances. In the ideal case, the variance is constant at zero during the entire simulation time. If asynchronous behaviour is added (lower curve), the variance increases as the nodes measure at slightly different points in time. By adding dependency (topmost curve), the logging of the signal is influenced by the actions taken at another node. The variance explodes and the system is in an entirely uncoordinated state.

7.2.1 Higher-Order States

Higher order states emerge through node interaction. For example, if node A with state S_a communicates with node B having state S_b they share a common composite state S_{ab} consisting of the conjunction of S_a and S_b. Composite state space increases with the degree of interaction among nodes. A system is considered more complex if the interaction among nodes is intense and hence the outcome of an action taken at node A which is interacting with k other nodes is difficult to predict. The degree to which an action taken by node A affects node B can be determined by the covariance. It is measured:

$$cov(A, B) = \frac{1}{n-1} \sum_{i}^{n} (a_i - \overline{a}) * (b_i - \overline{b}) \tag{7.1}$$

where A and B are discrete random variables and $\overline{a}$ is the average over the state history. A and B are sampled by recording states of node A and B. Analysing the entire system at time t yields a covariance matrix $COV(S_t)$. The Frobenius norm, $\|COV(S_t)\|_2$, provides a measure of higher order complexity. Figure 7.2a shows the covariances for the example.



(a) Covariance (b) Entropy

Figure 7.2: Covariance and entropy

As expected, the covariance value for the dependency case is much higher than for the ideal and asynchronous case which are almost the same. The covariance value, however, depends on the unit of measured values. Hence, the raw covariance value determines only whether there is a dependency (positive or negative value) or not (zero value). To compare two values the Pearson coefficient is used:

$$\varrho(X,Y)\frac{Cov(X,Y)}{\sqrt{(VAR(X)}\cdot\sqrt{VAR(Y)}} \tag{7.2}$$

7.2.2 Entropy

Another factor increasing the complexity of large distributed systems is the difficulty of determining the outcome of a particular action committed. Thereby, the predictability becomes worse with increasing entropy or randomness in the system. Classically, the entropy H of a system can be measured according to

$$H = \sum_{i=1}^{k} p_i log(\frac{1}{p_i}) \tag{7.3}$$

where $p_1 ... p_k$ is the set of probabilities for the k states the system can be in. Thus, if the system can be only in a small number of states each with high probability, the entropy is small, whereas systems with large numbers of states with low probabilities have high entropy and hence the predictability of an outcome to a corresponding action is more difficult.

Figure 7.2b plots the entropy for the example above. Although in all three cases the nodes have the same number of states, the entropy differs considerably. In the asynchronous and

feedback case the probabilities are lower, hence making the outcome of an action taken less predictable. In the ideal case the entropy is zero which is reasonable since all nodes are in perfect sync and hence the outcome of an action is based on a complete known state and therefore predictable.

7.2.3 Summary

The complexity of a system at a given time t is described using three metrics namely: the *variance* of states currently in the system, the *covariance* of state changes, i.e. the influence an action has on other nodes and the *entropy* inherent in the system providing a measure on the predictability of an outcome of an action. In the following sections these metrics are applied to evaluate how algorithms as well as architectural artefacts influence complexity. Additionally, the effort required to control the system is evaluated at several layers of abstraction, e.g. for the implementation of node modules or programs written in the query language. The rationale is that a programmer in a high level language needs to pay attention to fewer states since the underlying middleware provides the basic functionality for, e.g. communication or data management, hence effort required to achieve correct and robust programs is lower in the high level language.

7.3 Nodes

The five module node architecture has been qualitatively evaluated in Section 7.1. This section continues the evaluation by choosing key elements critical for the achievement of quality attributes. The quantitative investigation starts with the declarative monitor operator MON_k, and continues with analytical and simulative analysis of the query execution system with particular focus on the various facets of complexity.

7.3.1 The MON_k Operator

In Section 6.3.3 the ability of local node discovery was introduced. This section evaluates this capability using MON_k [51], a declarative operator to monitor groups of sensors, detect anomalies, and cluster nodes according to their semantic proximity. This section proceeds by presenting a foundation of the operator, i.e. a distributed k-means variant, followed by two applications, i.e. outlier detection and hierarchical clustering.

MON_k is designed to extract relevant information from a large set of globally distributed data sources. In the context of electricity networks it can be used to monitor high voltage

Listing 7.1: MON_k Definition

```
MON_k([COLUMNNS], [CORRELATION_FUNCTION], [DISTANCE], [NEIGHBOURS], [INITIALIZATION])

COLUMNS:              A set of column names to monitor
CORRELATION_FUNCTION: The correlation function
DISTANCE:             The maximum distance a node may have from the centroid
NEIGHBOURS:           The number of neighbours to exchange with in each iteration
INITIALIZATION:       A set of initial centroids
```

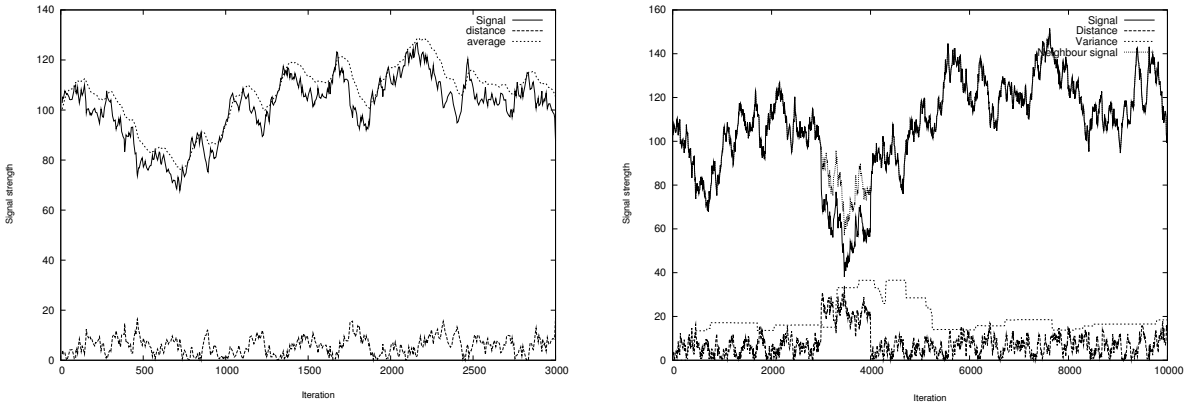
power lines or transformers in substations. With the information extracted by MON_k, relays or other control equipment can react to fault situations or take action to optimise network configurations. The operator executes similarly to the basic algorithm in Listing 6.1: It first partitions a set of given nodes into groups or clusters. Group membership is determined by some correlation function. Correlation functions can be as simple as the mean of a measured phenomenon or, e.g. more complex, the interplay of several different phenomena, e.g. voltage and temperature over a given time interval. Within groups, nodes gossip information with respect to the change of the measured phenomenon. The gossiping of information is implemented by the monitoring component which retrieves information from nodes contained in the local nodes table as described in Section 6.3.5. Using a distance metric, each node determines its deviation from the common group state, e.g. distance of own measurement to the mean over all measurements within its group. Based on this local computation it is decided whether the node belongs to the query result set `R` like specified in Query 7.2 (lines 5-6). In the listing, a subset of nodes is selected and fed to the MON_k operator which uses the average of the measured phenomena as correlation function and the Euclidian distance to determine its deviation from the stable state.

Listing 7.2: MON_k Query Example

```
1 N -> SELECT *
2         FROM SENSORS
3        WHERE type='voltage'
4
5 R -> SELECT MON_k(voltage, AVG, DIST, 10, {1, 101})
6          FROM N
```

The operator initialises at each node by computing the minimal distance to the given initial means ({1,101}). It then chooses 10 neighbours at random. Collecting their measured values and applying the average function, the initial average is updated. Neighbours with strong deviations from the average are replaced by randomly chosen nodes from the supplied set.

An experiment demonstrates the capabilities of MON_k. We consider a set of nodes that measure a Gaussian signal (Figure 7.3). The initialisation process and the convergence



(a) MON_k Initialisation phase

(b) MON_k Failure detection

Figure 7.3: Outlier detection at runtime. Simulation with 5000 nodes;k=10 neighbours

of the mean is depicted in Figure 7.3a. The topmost curve, representing the mean, follows the signal quite nicely with a small delay. The delay is due to the time required for the mean to converge. It can be reduced by increasing the intensity of gossiping between nodes. The bottom curve shows the distance from the mean as measured by one randomly chosen node. In an ideal case, the distance should remain almost constant i.e. a horizontal line. However, as elaborated in [5] and [15], stability of the algorithm depends on various conditions of the execution environment. Therefore the simulation was done using realistic network models with varying transmission latencies. Furthermore we do not assume synchronised clocks at the nodes and MON_K is capable to handle failure situations like failing nodes and message loss.

The MON_K allows to cluster nodes according to a correlation function. MON_K can be extended to build a hierarchy of clusters. Therefor, the algorithm is executed as elaborated above. Once clusters begin to converge, for each cluster a cluster head is selected. To accomplish this, nodes of the same cluster exchange their distance to the cluster centroid with the nodes in their nodes table. The one that is closest to the cluster centroid becomes the cluster head. In the case of equal distances two cluster heads may be chosen. This has no influence on the stability of the algorithm. Once the cluster head is determined the process is complete for the first level of the hierarchy. In the next round cluster heads engage in the gossip to generate clusters of the next layer with an increased cluster radius. The process is continued until the largest cluster radius has been processed. This hierarchical clustering method can be used to cluster nodes in order to implement a remote backup protection system as introduced in the corresponding scenario in Chapter 6.

Based on data from the European transmission network, UCTE, the hierarchical version

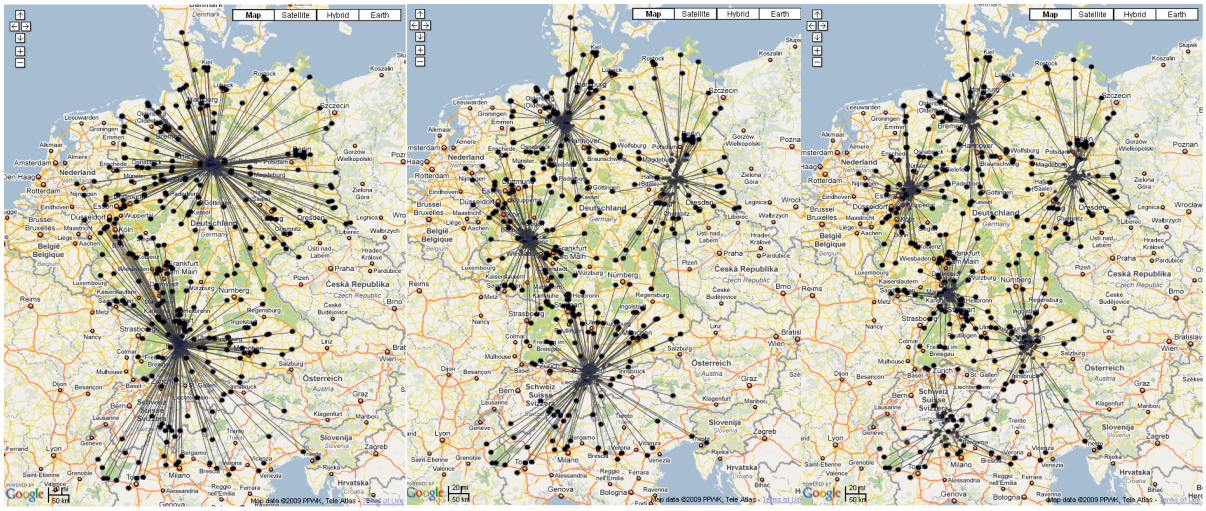


Figure 7.4: Different cluster layers

of MON_k has been simulated. In an experiment MON_k was set up to cluster all electrical substations in Germany. Simulations involved 147 substations from the four major electric utilities, i.e. EON, EnBW RWE and Vattenfall.

Correspondingly, Figure 7.4 illustrates simulation results. The figure shows exemplary substation clusters at three different layers. Clusters emerge after a few iterations and stabilise quickly initiating the formation process at the next layer. Besides the dynamics of the simulated communication network, substation information is static. Hence simulations support the assumption of a stable and reliable overlay. Geographic information from substations can be considered static with a high reliability of devices. Therefore, after the initialisation phase completed, discovery queries can be set to long periods of inactivity or transformed to subscriptions.

Clustering and grouping of nodes is an important capability of the node architecture. Using MON_k enables content based clustering of nodes. Grouped nodes increase the intensity of communication as the monitoring module automatically monitors the nodes in the local node table. Hence, detailed and up to date information about neighbouring nodes are available. This property can also be utilised to implement highly available execution of queries and programs as elaborated in [50], [52], [38] and [110].

7.3.2 Query Execution

The query optimisation process is critical in order to fulfil quality of service constraints and make effective use of available resources. Depending on the type of query, effort required for optimisation is considerable. In particular aggregation queries constitute a complex

problem as the search space, consisting of alternative aggregation trees is large $O(N!)$. The problem of finding the optimal query configuration falls in the class of NP-Hard problems.

Lemma 7.3.1 *Optimisation of aggregation queries is NP-Hard.*

Proof Consider k sources $S_0...S_k$, i nodes $A_0...A_i$ that can be used to compute an aggregation and a receiver R. Let the graph $G = (V, E)$ with nodes V contain all nodes and sources and edges E and let the edges of nodes be able to directly communicate with each other. The aggregation tree with the sources at the leaves and the receiver of the root is the reverse of a multicast tree. A multicast tree with the minimum number of edges is a minimum Steiner tree on the network graph G. Hence, assuming an arbitrary placement of sources and a general graph G, finding the optimal aggregation points is NP-Hard [79]. ∎

For very small queries the optimisation overhead is negligible. More complex queries such as the one depicted in Figure 7.5 require considerable time to find the optimal configuration. The example query requires already 45 seconds to optimise for six resource candidates on an Intel Core Duo 2 computer with 2.8 Ghz. On the same machine it takes almost an hour to optimise for 10 execution candidates.

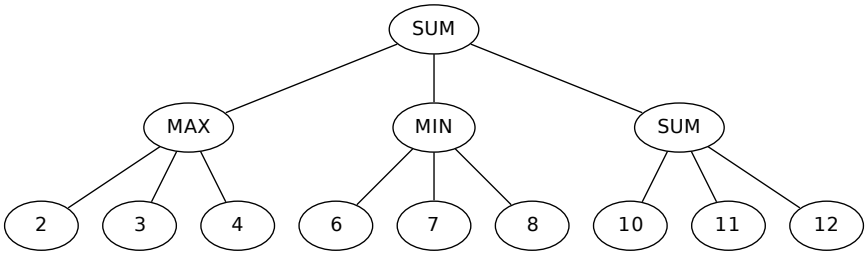
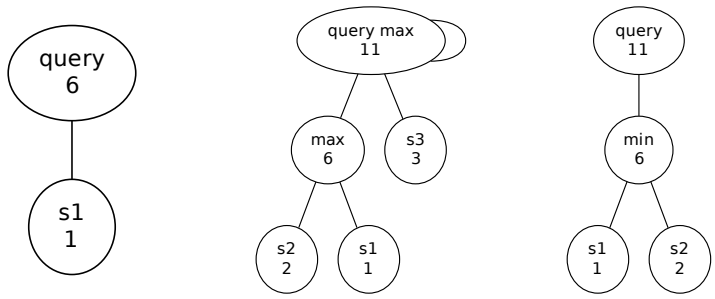


Figure 7.5: Example query taking considerable effort to optimise

This behaviour is clearly not acceptable and must be anticipated. In the last 30 years a considerable body of research has been conducted on query optimisation for local and distributed problems yielding excellent results with heuristic approaches. In the open design of the query optimiser additional optimisation methods can be implemented. Heuristic query optimisation itself is outside the scope of this work, it is at this point referred to the related literature [3], [101] and [151]. Another aspect of aggregation query optimisation, however, is covered in the next paragraph.



(a) Query A (b) Query B (c) Query C

Figure 7.6: Multi-query optimisation

Multi-Query Optimisation

In this section the ability to optimise multiple queries is analysed using an example scenario. Considered are the three example queries from Section 6.3.3, however, with a slightly different initial placement as shown in Figure 7.6.

Implementing the cost based optimisation approach, queries are optimised for data reuse if the optimisation constraint includes the cost of data retrieval. Two queries processing the same data sources can share the data contained in memory and hence save redundant transmission of data and re-processing of intermediate results such as aggregated values. Since optimisation is applied iteratively for all queries currently in execution, queries processing the same data will be clustered in the same nodes or reconfigured such that sub-queries cluster at the same nodes. For aggregation trees with depth ≥ 2 result sharing and early aggregation may produce sub-optimal results as illustrated by the following example based on the three queries in Figure 7.6.

The optimiser determines the minimum cost for each query. Hence, for query B, it decided to place the aggregation of the measurements of sensors 1 and 2 at node 6 while retrieving the measurement from sensor 3 directly for the final aggregation at node 11. Although individual optimisation yields optimal results for each query, in the context of multiple queries the result is sub-optimal. Although one aim of aggregation is to reduce transmissions over the network it may cause additional traffic. While in a multi-query context, results are shared among queries, aggregation functions compress information thereby generating a query specific result which cannot be shared with other queries. In the above example this becomes evident by counting the messages transmitted for each query execution cycle. By comparing the results for different placements with or without aggregation

	Query A	Query B	Query C	$\sum$
00	1.5	-	-	1.5
03	1.5	-	-	1.5
05	-	4.5	-	4.5
06	2.5	-	-	2.5
09	4/3	-	-	4/3
10	-	13/3	-	13/3
12	4/3	-	-	4/3
15	1.2	44/15	-	62/15
16	-	-	43/30	43/30
18	1.2	-	43/30	79/30
20	-	46/15	47/30	139/30
22	-	-	5/3	5/3
24	-	-	5/3	5/3
25	-	19/6	-	19/6
26	-	-	5/3	5/3
28	-	-	2.5	2.5
30	-	-	2.5	2.5
$\sum$	317/30	18	433/30	43

Table 7.2: Relative costs per transmission with aggregation disabled

the aggregation efficiency can be measured. Tables 7.2 and 7.3 summarise the results[5]. Without aggregation, i.e. individual transferral of data items to the sink nodes, a total of 43 messages is transmitted. Interestingly, with aggregation enabled, i.e. compression on the first level possible (nodes 6 and 11) a total of 44 messages are transmitted.

While the example shows the complexity arising in a multi-query context, a careful set-up of the cache weight in the optimisation criteria will lower cost to placements with fewer aggregations and hence yield the best fitting solution during iterative optimisation of the query. Also the query optimiser might utilise heuristic or evolutionary approaches to find the optimal configuration in the long run.

[5]To calculate the costs per query in the case of shared result sets, equal fractions are counted (e.g. if one message transports result set for three queries, 1/3 is added to the cost of each query).

	Query A	Query B	Query C	$\sum$
00	1.5	-	-	1.5
03	1.5	-	-	1.5
05	-	4	-	4
06	2.5	-	-	2.5
09	4/3	-	-	4/3
10	-	23/6	-	23/6
12	4/3	-	-	4/3
15	1.2	3.4	-	4.6
16	-	-	1.4	1.4
18	1.2	-	1.4	2.6
20	-	53/15	38/15	91/15
22	-	-	31/12	31/12
24	-	-	19/12	19/12
25	-	43/12	-	43/12
26	-	-	19/12	19/12
28	-	-	2	2
30	-	-	2	2
$\sum$	317/30	18.35	181/12	44

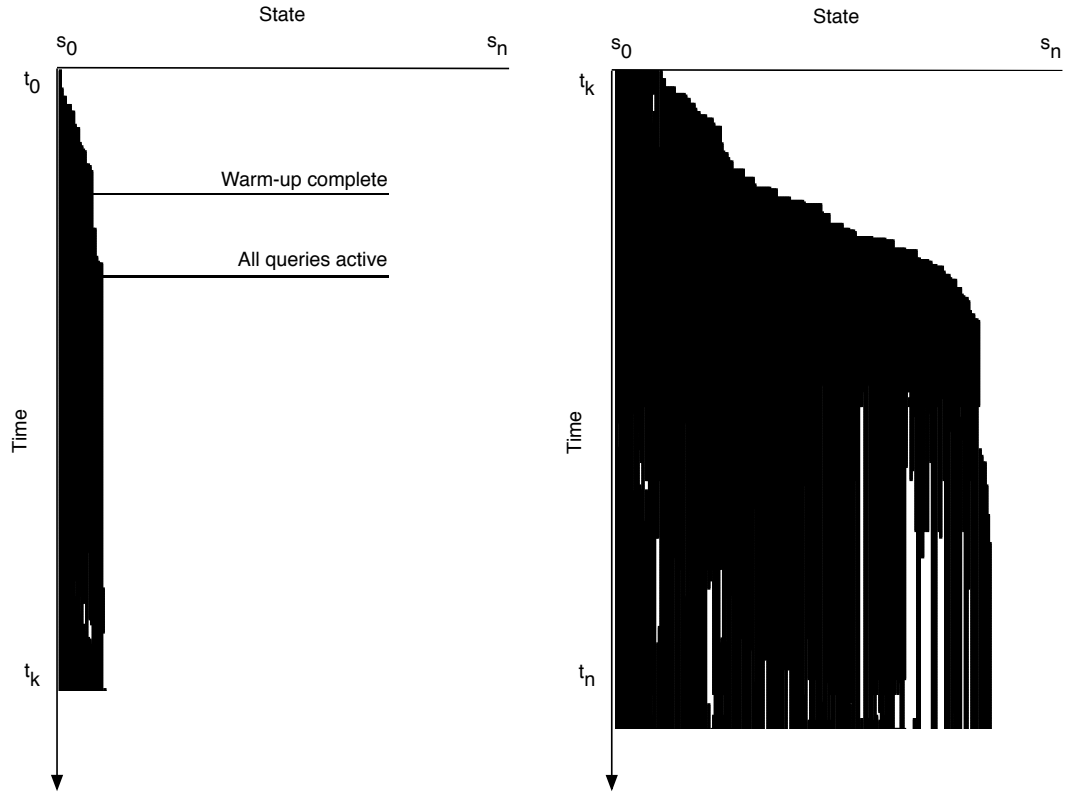
Table 7.3: Relative costs per transmission with aggregation enabled

Complexity

The network of nodes may exhibit complex behaviour which may yield inefficient resource allocation and even endanger the stability of the entire system. Being limited to partial knowledge about the system state, query processors can only optimise the local resource utilisation, i.e. its own resources plus resources of neighbouring nodes. Figure 7.7 shows the outcome of an experiment where the three types of simple queries (Figure 7.6) from the previous section have been injected in a network with 200 nodes. The optimiser is set to produce query configurations such that resource (computation) consumption is optimised among nodes. For matters of simplicity, all nodes have just a single resource of uniform capacity with the load dynamics pictured in Figure 7.8. The curve is the result of benchmarking a single node hosted on a Core Duo 2 Intel system with 4 GB of RAM running Windows XP. The test load consisted of the queries in Figure 7.6 executed as snapshot queries. Up to 6000 concurrent clients, i.e. injection, execution and result set delivery, where simulated. During the experiment the same queries are executed as continuous queries. To observe the behaviour independent of external events, query injection happens within an initial warmup phase at which end (after 1000 iterations) all queries are scheduled.

Figure 7.7 illustrates the memory of a randomly chosen node in the experiment. The figure can be interpreted as follows: since the memory module is the only component able to store information between compute cycles (s. Section 6.3.2) it contains the entire node state. Assuming that the memory has unlimited capacity, each microstate, i.e. item stored to a memory cell, can be identified by a unique position (an integer value). For example, the communication module might be in the middle of the execution of an stabilisation protocol. A node might wait for a defined time interval until a response arrives, hence it stores the current protocol state together with a time-out value in memory. This information is assigned a unique and consecutive position specifically for this particular information. In the sequel, memory cells currently storing information are referred to as active, while empty cells are non active. Figure 7.7 shows the state evolution by depicting active microstates with a black dot and not active microstates with a white dot. The sum of all microstates at the same horizontal level represent the complete state of the node at a given point in time. The state evolution starts with the top row and proceeds gradually towards the bottom.

The experiment shows two characteristics very common in large distributed systems. After the warmup phase is complete and all queries are active, the system seems to be in a stable and predictable state (low entropy). After some time and without external stimuli the dynamics changes abruptly. Thus, generalising from the observations, the first charac-



(a) Initial cache evolution

(b) State explosion long time after the last query became active

Figure 7.7: State evolution as measured at a single node in an experiment with 200 nodes and constant query load. Illustrated is the content of the memory of a randomly chosen node in the experiment. Each microstate, i.e. item stored to a memory cell, can be identifed by a unique position (an integer value, x-axis). Active memory cells are depicted as black dot and not active cells with a white dot. The state evolution starts with the top row and proceeds gradually towards the bottom.

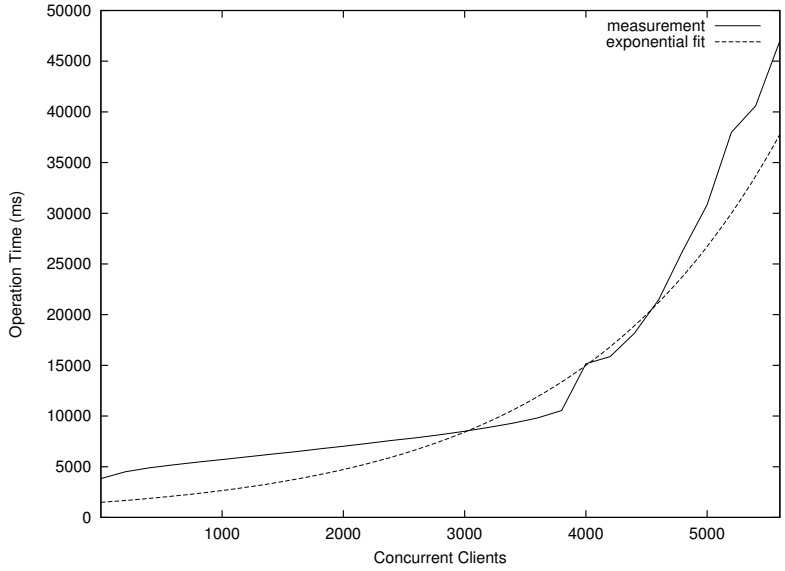


Figure 7.8: Node resource dynamics

teristic is that effects of actions taken may become apparent after considerable amounts of time. In the Peer-to-Peer and Grid Computing lab at Siemens Corporate Technology this phenomenon has been observed in many highly distributed systems. Effects can be delayed by hours or even days. The fact makes failure analysis and debugging extremely difficult, as failure situations are hard to reproduce and the failure root causes are extremely difficult to identify. The second characteristic is that once the system state changes it does so rapidly and without many prior indications. State variations at one node affect the state at other nodes, causing a chain reaction which has the potential to render the entire system unusable. Both characteristics are implicit and it is clear that they cannot be controlled easily as the state evolution depends on many only partially known and randomly occurring stimuli.

In correspondence to the visualisation of the state evolution in memory (Figure 7.7), Figure 7.9 shows the entropy measured during the same experiment. Shown are the effects with the stabilisation mechanisms both enabled and disabled. Without stabilisation, entropy increases rapidly (topmost curve). In this situation about 90% of effort on a node is spent for reallocation of resources. After the initial increase, entropy oscillates with large amplitudes. In this state the system is not only inefficient; its behaviour becomes also hard to predict, which directly influences the optimisers in other nodes which might try to also allocate resources on other nodes at the same time. Since their assessment of their resource utilisation is most likely not accurate, their optimisation output further

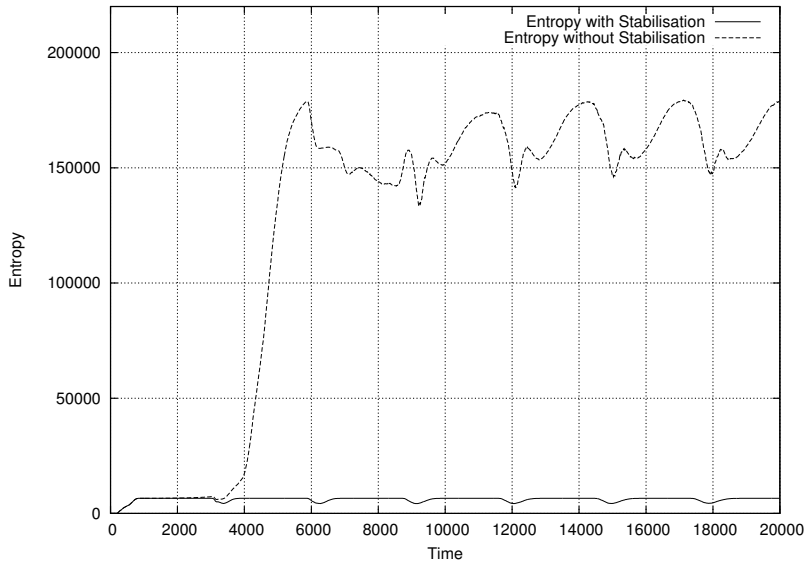


Figure 7.9: Nodes with disabled stabilisation enter an uncontrolled state

contributes to the reallocation overhead. On the contrary, with stabilisation mechanism applied, i.e. including entropy costs in the optimisation process, the entropy remains almost constant in a bounded condition with slight periodic oscillations as shown by the lower curve in Figure 7.9.

7.4 Index Cloud

Additionally to the standard query processing functions, index nodes offer additional functionality to handle large data sets consistently. This section deals with the data management capabilities of the index cloud. It uses the simulation methods described in Section 4.3 to evaluate the algorithms presented in Chapter 6. The basis for the simulations constitutes a prototypical implementation of the node architecture and corresponding data management algorithms. The node kernel was replaced by a simulation kernel (s. Section 4.3). Additionally the communication module is replaced by its simulation counterpart which is able to load different network models. Unless otherwise stated, a realistic network model [78] is used to provide a real-world environment.

Table 7.4: Failure induced write load

Nodes	Writes Per Second (MTTF 30 minutes)	Writes Per Second (MTTF 24 hours)	Writes (MTTF 10 years)
1000	0.56	0.01	2 / week
10000	5.56	0.12	3 / day
100000	55.56	0.16	1 / hour
1000000	555.56	11.57	11 / hour
10000000	5555.56	115.74	2 / min
100000000	55555.56	1157.41	19 / min

7.4.1 Performance

The index cloud has considerable influence on system performance. It is optimised for read operations and capable of delivering timely information for local query processing. However, with increasing system size, load patters shift from mainly read to mostly write operations. Assuming that the index cloud is used primarily as index for node meta data, writes occur only when new nodes arrive, i.e. new devices are installed, or nodes reregister due to a failure situation, e.g. system failure or network interruption. In the following a model for read and write workloads is elaborated. Based on this model structure and algorithms of the index cloud are evaluated.

Table 7.4 summarises write loads in correspondence to the number of nodes in the system and an assumed mean time to failure (MTTF). While MTTFs of 30 minutes are common for nodes in the Internet [141], MTTF in the range of years are more likely for devices in an industrial environment. Although, due to the lack of a broad installed base, no data is available for metering equipment at end-users, a MTTF of 24 hours is assumed for the following considerations. The assumption is supported by the fact that most DSL providers reassign IP addresses on a 24h base. Moreover, for the further analysis it is assumed that consumer devices constitute the majority of all participating nodes and hence an MTTF of 24h as lower bound is the base for all further calculations.

In addition to the write operations triggered by system or network failures, soft state data items need to be renewed to prevent their deletion. The renew operation is less complex since it targets only non persistent data and does not require indexing[6]. However, considering a renewal time of 1 hour would yield already 3000 operations per second in a system with 10 million nodes and one data item per node (node meta data). For

[6]Since the data item is not updated and metadata as well as item content remain the same

certain applications, a renewal time of 1 hour might be too long, e.g. frequently executed queries over many nodes will require considerable rewrite overhead until the failed node is removed from the nodes tables. The monitoring system introduced in Section 6.3.5, however, improves this process by quickly detecting node failures and by using the failure hook mechanisms to clear outdated data promptly. Hence, renewal times can be set to higher values. A renewal time of a week would reduce the number of renewal requests to 17 per second. Assuming 10 monitoring nodes and an MTTF of 24 hours the number of updates by failure hooks is in the worst case $10 \times 115 = 1150$ per second whereas only the update by the first monitoring node has influence on performance because the data item will be removed and further updates will fail. Taking this into consideration the administrative write load $Load_w$ on the index cloud in a system with 10 million nodes is $115 + 115 + 17 = 247$ updates per second.

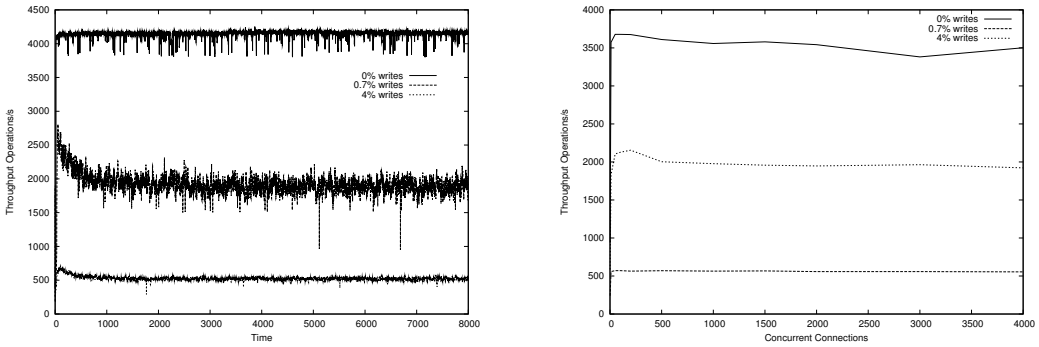
Estimating the read load on the index cloud is difficult as it depends on the types of devices currently active in the system as well as applications using the system. To provide a rough estimate it is assumed that data is read from each device, i.e. at least one query per node exists. In general, data needs to be read from the global nodes table every time a declarative query is executed unless meta data is cached in the local nodes table[7] and no new execution candidates need to be found. Considering a continuous query for every node in the system and an optimisation interval of 5 minutes to find new execution candidates yields read load of 33333 read operations per second in a 10 million node system or in other words 10 million concurrent queries. The resulting read/write ratio is $247/33333 = 0.007$[8].

To find new candidates for query execution, however, a subscription could be more efficient. In the context of the scenario above, the throughput of subscriptions that need to be fired equals the number of writes plus the effort to evaluate all subscriptions for the update plus overhead to renew the subscription. The number of updates that need to be transferred to clients depends on the query structure. A query selecting a large number of rows is more expensive than queries with smaller result sets. Hence the outgoing load for subscriptions is $L_{so} = S_r i \times Load_w$ with S_r the size of the results set for subscription i. Hence total subscription load is $S_l = L_w + L_{sq} + L_{so}$.

Independently of the consistency mode, the write performance of fully replicated clouds is limited by the maximum number of write operations supported by the slowest index node since eventually all updates need to be written at any node. Depending on their config-

[7]State information is not part of the global index but is retrieved by the monitoring module during execution

[8]Numbers are based on a Siemens internal report on the load patterns of corporate enterprise applications, e.g. wikis, blogs, social networks



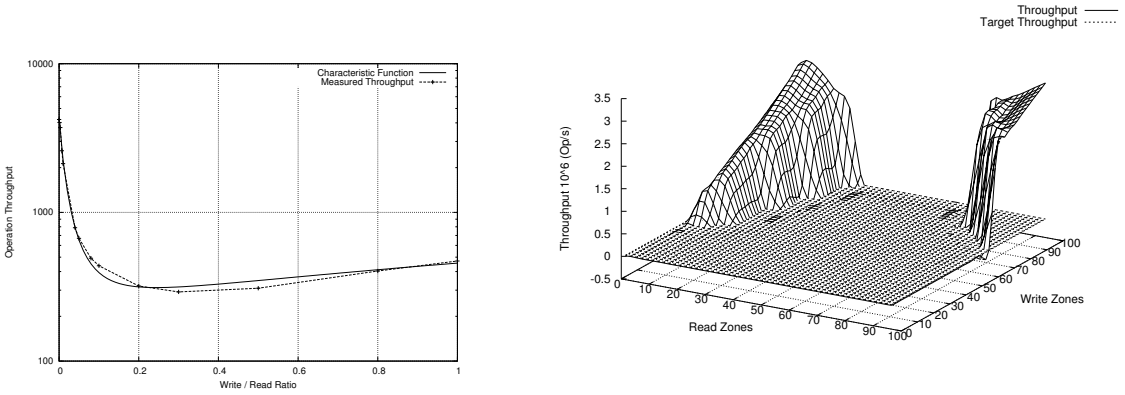
(a) Throughput over time (b) Throughput vs. concurrent Connections

Figure 7.10: Index node operation throughput

uration, partitioned clouds could yield better performances in the context of high write load requirements. To provide a rough estimate on read and write throughput, an index node has been implemented and extensively tested. The testbed consisted of two Apple Laptop Computers with Intel Core 2 Duo CPUs clocked at 2 GHz, 2 GB of RAM and connected via 1000baseT ethernet. In a base test, the behaviour of the node with regard to different write-read ratio was tested. Therefor, the client computer created a table on the index and filled it with 10000 rows and generated 100000 queries. Generated at random, queries requested rows by specifying five conjunctive conditions, i.e. "attributeA=B AND attributeB=C AND ...". Subsequently, up to 4000 concurrent client connections[9] were simulated and each client issued read and write operations on the index node in the ratio elaborated above. The index node implements a simple caching scheme where query results are cached by the query and invalidated once an item matching the query is written. For all simulations time measured is net execution time at the server without query compilation and transfer over the network. Since performance greatly depends on the implementation efficiency, hardware, network and test data set, experiments were not designed to measure peak throughput rates. Rather, the behaviour, in particularly with variation of different write-read rations, shall be investigated.

Figure 7.10a shows the throughput for 100 concurrent connections over approximately 2 hours time averaged over three sequential runs. The figure shows that in the absence of write operations caches heat up quickly and throughput settles at around 4100 operations per second. With 0.7% of all operations being writes, the node starts with a throughput around 2500 operations per second due to the initial writes of the 10000 rows. Over time throughput converges to a rate around 2000 operations per second. For 4% write

[9]The number of concurrent connections was limited by a bug (id 6932633) in the OSX ethernet driver on the client side.



(a) Throughput vs. write-read ratios (b) Throughput vs. partitioned cloud configuration

Figure 7.11: Cloud configurations for varying write-read ratios

operations a similar initial behaviour is observed as for the previous case. Eventually, the rate settles at around 500 operations per second. As Figure 7.10b shows, rates are stable in the context of varying concurrent connections. To speed up the test, fewer operations (approximately 400000) were executed per write-read ratio, hence the slightly lower throughput for the 0% writes case. All experiments were conducted without any optimisations to network, operating system or sophisticated caching mechanisms at the index node. Tests with higher read percentages were limited by the client performance as the index node was utilised below 50%. Tests with many writes prevented efficient caching and hence increased the load on the index node.

The read and write performance of a partitioned cloud can be controlled by increasing or decreasing write or read zones. In the following a method to find the optimal configuration for a given operation load (read-write ratio) is developed. Figure 7.11a shows total cloud throughput for different write-read ratios measured using the same set-up as detailed above and executing 60000 operations during each run. A ratio of 0 means only read operations, 0.5 means 50% each and 1 means only write operations. Clearly, the performance curve depicted is specific for the particular index node implementation, caching scheme, the operating system, the hardware and network infrastructure. To achieve the performance requirements elaborated in previous paragraphs (33333 read and 247 write operations/s) the sum of individual node throughputs must match the required load. Equation 7.4 models the node behaviour with $T(n)$ the throughput for index node n under given conditions. To find $T(n)$ the measured curve in Figure 7.11a is approximated (using a standard interpolation algorithm [113]) with Equation 7.5.

$$readLoad + writeLoad = \sum_i T(n) \tag{7.4}$$

$$T(x) = \frac{c_0}{(1 + \frac{x}{v_0})^m} + \frac{b}{a} \times (x + 1) \tag{7.5}$$

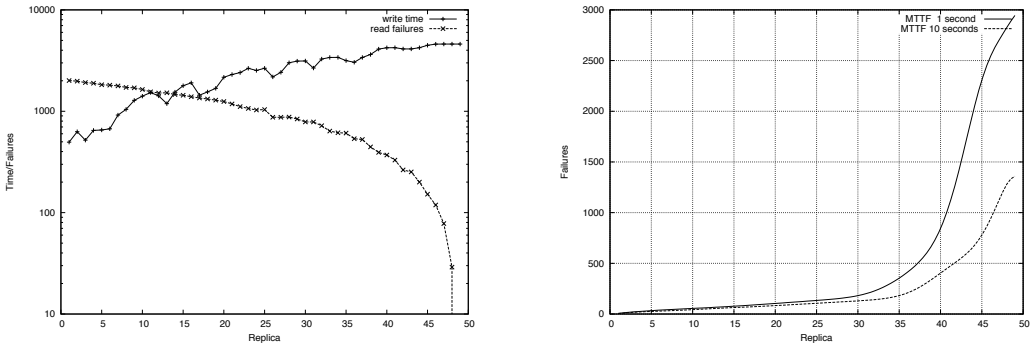
with $c0 = 4110.14$, $v_0 = 0.024$, $m = 2.04172$, $b = 500.113$, $a = 2.20358$ (Figure 7.11a). With the characteristic function known, equation 7.4 can be solved. Plotting all configurations for different read and write zones, Figure 7.11b shows all solutions that meet the requirements as surface above the target plane (x,y, 33580).

Automating the method described above and feeding the configuration information back to the index master allows for autonomous organisation of the cloud. The monitoring module determines the base for the characteristic function and the master fits the curve and computes the optimal number of read and write zones. Thereby not only the configuration effort is minimised, also the cloud can adapt to changing load patterns by adjusting the number of read and write zones. In the extreme case, the partitioned cloud degenerates to a fully replicated cloud to achieve maximum read performance, i.e. $wz = rz = 1 \rightarrow Throughput_{read} = n \times Throughput_{node}$, with n being the number of index nodes. On the other extreme, write performance is maximised with $rz = 1, wz = n \rightarrow Throughput_{write} = n \times Throughput_{node}$. Reconfiguration effort, however, is considerable such that load pattern detection must be stable.

7.4.2 General Behaviour

While in the previous section experiments on a concrete testbed gave insights into how the index cloud must be configured for various load demands, in this section the general behaviour of the index cloud with varying consistency properties is analysed. To be able to conduct experiments with large numbers of nodes, simulations were executed on the distributed discrete event simulator of Chapter 4. As network model the Kings data set [56] was used. The simulation time frame (tic) equivalents one millisecond real-time. To simulate limited node compute and network resources, utilisation was modelled with $ce^{\frac{\sum_{t-t_k} O-c}{5}}$, with c, being the capacity of the resource and $\sum_{t-t_k} O$ the number of resource uses in the time period $t - t_k$.

The index cloud behaviour is massively influenced by the different consistency models. While strong consistency ensures that when an item is written subsequent reads always access the newest version, eventual consistency allows a time window where a read subsequent to a write may not return the latest version. From a client perspective eventual consistency writes are faster than strong consistency writes. Inconsistent reads, however,

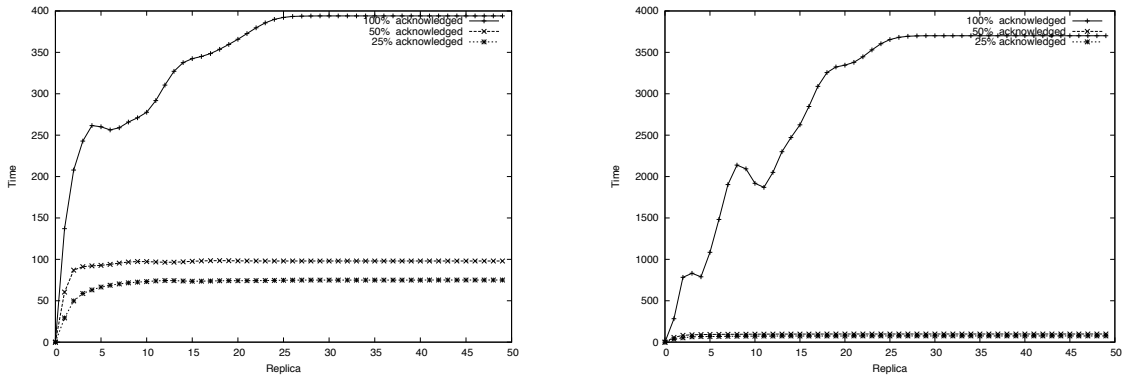


(a) Write time vs. read failures (b) Write failures vs. number of replicas

Figure 7.12: Stability of the write operation

may require additional failure handling at the client. Figure 7.12a shows the relation between time required for a write operation for different numbers of acknowledges required for a successful write and number of read failures. In the simulation an average online time for index nodes was set to 30 minutes. Moreoever a write load of 10 writes per second and constant read load of 100 reads per second was set. A read failure is defined as an item that is read and has a version number smaller than the latest item written with the same ID. As depicted in Figure 7.12a, writes requiring few acknowledges are considerably faster than writes requiring more acknowledges. Additionally, eventual consistency write performance is less dependent on the number of replicas. Failures increase inversely proportional with the write times. This behaviour complies exactly with the definition of strong consistency. In the strong consistency model, write performance is constraint by the performance of the slowest replica, while eventual consistency write performance solely depends on the k fastest replicas.

Performance of the write operation is also bound to the reliability of replicas. Especially in the strong consistency case a low mean time to failure (MTTF) can cause considerable delay. Figure 7.13a-b shows the write performance of successful write operations for a MTTF of 1 minute and 30 minutes. The write operation is not only slower in average but the number of write failures increases with the number of replicas (Figure 7.13b). Consequently, strong consistency might not be achievable in large index clouds when the average MTTF drops below a certain threshold. This is shown in Figure 7.12 where very short MTTF values were simulated. Failures increase exponentially with the number of replicas rendering successful writes impossible. On the contrary, eventual consistency write operations are unaffected by the total size of the cloud and are more reliable to node failure (s. Figure 7.12a-b) since, when k acknowledges are needed, $n - k$ out of n replicas



(a) MTTF 30 minutes

(b) MTTF 1 minute

Figure 7.13: Performance of the write operation

can fail before the write operation as a whole fails[10].

Read operations in fully replicated mode are executed on a single node that received the request. Stability of the read operation in fully replicated mode is therefore dependent solely on the availability of one index node. In partitioned node, a read request is distributed to all write zones within a read zone to retrieve partial results. The operation is similar to the write operation as more index nodes need to be contacted and the read will fail if one node handling a sub query fails. Therefore, the failure probability of the read operation in partitioned mode increases proportionally with the size of the read zone similarly to the write failures depicted in Figure 7.12b. Independent of the operating mode, the read operation depends on the availability of individual index nodes. The availability and its influence on the data cloud is further analysed in the following section.

7.4.3 Availability

The availability of index nodes has considerable influence on the index cloud. As elaborated in the previous section the success of read and write operations depends on the availability of individual nodes. Besides analysing the availability of index nodes, this section investigates the availability of stored data, since in case of failure of large numbers of nodes even replicated data might get lost.

Assuming equal mean time between failures (MTBF), the failure of individual nodes being independent of other nodes and a service time t_s for a read or write operation, then the probability that an operation fails is:

[10]This assumes equal MTTF for all nodes in the index cloud.

$$P_f = \frac{MTBF}{t_s} \tag{7.6}$$

Concretely, for a MTBF of 24 hours and $t_s = 400ms$ and considering that the operation affects a single node only, the probability is 4.63×10^{-6}. If the operation includes several nodes, the probability for node failure is the sum of individual probabilities, i.e. the probability of a strong consistent write failure in a large cloud with 1000 nodes is $1000 \times 4,64 \times 10^{-6} = 4,64 \times 10^{-3}$. Hence, with the exception that number of write zones = 1 and read zones = 1, strong consistent writes in partitioned mode are more reliable than in fully replicated mode when considering the same number of index nodes. However, since in fully replicated mode the number of replicas is generally larger, data availability is higher in the fully replicated scenario. Given n index nodes and data which is replicated to r replicas including the node responsible for storing the item, each having an average availability of A_i the probability that data is lost P_{loss} is:

$$P_{loss} = (1 - A_i)^r \tag{7.7}$$

Concretely, assuming an index size of 100 nodes with average availability of 99.9% and $r = 3$ replicas or three read zones P_{loss} is $(1 - 0.999)^3 = 1.00 \times 10^{-9}$. A fully replicated scenario yields P_{loss} is $(1 - 0.999)^{100} = 1.00 \times 10^{-9}10^{-44}$. Assuming an average host availability α_h for all nodes, Equation 7.8 determines the number of required replicas in order to achieve a minimum data availability.

$$A_d = 1 - (1 - \alpha_h)^r \tag{7.8}$$

$$r = \frac{log(1 - A_d)}{log(1 - \alpha_h)} \tag{7.9}$$

Using Equation 7.8, Table 7.5 shows the number of replicas needed to achieve a given data availability while assuming an average host availability of $\alpha_h = 0.8$.

In above considerations individual node failures are not correlated. In certain situations, e.g., blackouts or network failures, large numbers of nodes may fail at the same time. The question to answer in this scenario is how many nodes may fail at the same time until data availability can no longer be guaranteed? The recursive Formula 7.10 elaborated in [134] is used to answer this question.

Table 7.5: Number of replicas needed to achieve a given data availability. Average host availability is 0.8.

Target Availability	Replication Factor (r)
0.8	1
0.9	2
0.99	3
0.999	5
0.9999	6
0.99999	8

$$g(n,k,r) = \begin{cases} 0 & \text{if} \quad n < 0 \\ 1 & \text{if} \quad \begin{array}{l} n = k \text{ and} \\ 0 \leq n \leq 2 \end{array} \\ \sum_{i=0}^{r} g(n-i-1, k-i) & \text{otherwise} \end{cases} \tag{7.10}$$

The probability of data loss if k out of n nodes fail is $P_{loss} = 1 - \frac{g(n,k)}{m(n,k)}$ with $m(n,k) = \binom{n}{k}$, i.e. the number of variations to choose k nodes out of n and $g(n,k)$ the number of ways to choose k out of n failing nodes without data loss. Figure 7.14 shows the result for varying number of nodes and a replication factor of $r = 3$. The probability of the loss of an entire replication group decreases with the total number of nodes. Independently of the number of nodes, if $\frac{2}{3}$ of the nodes fail, the probability that at least one replication group fails is 1. Surely, in fully replicated mode all nodes must fail before any data is lost. Loosing data in partitioned mode is equivalent to losing the same write zone in all read zones. Assuming three read zones and no replication within a write zone Formula 7.10 can be applied. For replication factors other than 3, the formula can be extended as elaborated in [134].

A critical element for the availability of the index cloud in both partitioned as well as fully replicated mode is the index master. If the master is unavailable, new nodes cannot join and index nodes are not informed of failed nodes. In Chapter 6 it is suggested to chose a highly reliable hardware system as master. However, even the most reliable server systems have a probability for failure. To compensate such failures the architecture implements a watchdog mechanism. A master is monitored by a set of watchdogs which, if a failed master is detected, take over the master role. Given master and watchdogs have an availability

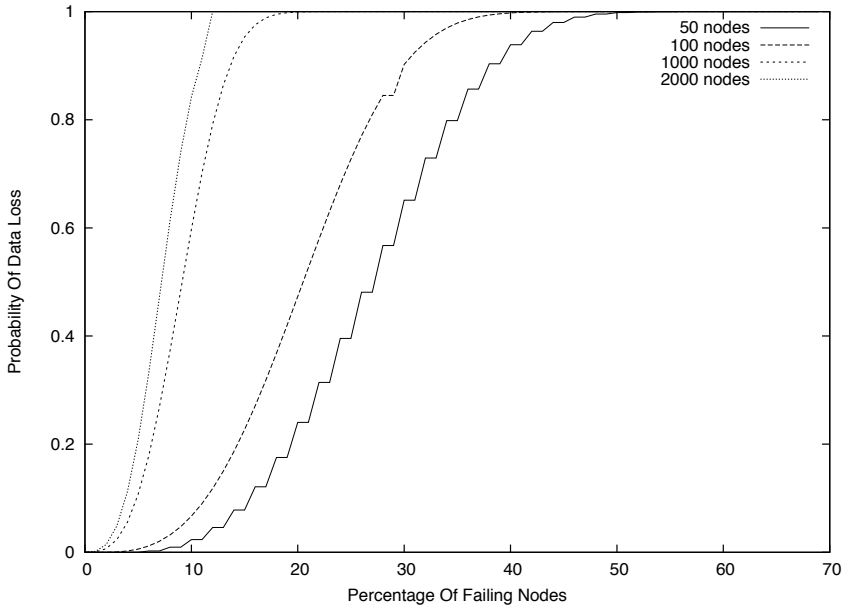


Figure 7.14: Data loss in the presence of massive node failure

of 99.999% the probability master and watchdogs fail simultaneously is $(1 - 0.99999)^{w+1}$, with w the number of watchdogs. Considering two watchdogs, this yields a probability of 10^{-15} or highly unlikely. Depending on the rate of probe messages sent from the watchdogs, in the event of a master failure, there will be a time window where no master is available. In order to switch to master mode, the effort for a watchdog is limited to the detection if a probe message remains without response. Hence, given a probe rate of 1 second and requiring 3 messages without response the system is without a master for 3 seconds in the worst case.

7.5 Programming Language

In the previous discussion on the achievement of quality attributes, the contributions of the language interface to modifiability, integrability and, implicitly, to security have been investigated. A pre-requisite of these contributions is the ability of the language to express all tasks and interactions within the ecosystem for energy services. Therefore, in this section, the expressiveness of the language is proven. Moreover, applicability and usability of SCSQL are shown by implementations of concrete control problems.

7.5.1 Completeness

Lemma 7.5.1 *The SCSQL is Turing complete.*

Proof The expressiveness of a query language is defined as the class of functions it can express on input tape [3]. In the following, it is shown that SCSQL can compute arbitrary functions encoded as Turing machine. A Turing machine is defined by the tuple $M = (Q, \Sigma, \Gamma, \delta, b, q_0, F)$ where Q is a finite set of states, $\Sigma \subseteq \Gamma \backslash \{b\}$ is the set of input symbols, Γ is a finite set of tape symbols, $b \in \Gamma$ is the blank symbol, $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function with L, left, encoded as 1 and, R, right, encoded as -1, $q_0 \in Q$ is the initial state and $F \in Q$ is a set of accepting or final states.

In SCSQL the Turing machine is defined in four tables: (i) a table called *state* keeping the current state, the current symbol and position on the tape, (ii) a table called *accept*, (iii) a table called *tape* containing the input and (iv) a *transition* table. Listing 7.3 shows the table definitions and Listing 7.4 the Turing machine processing logic.

The Turing machine works as follows: For each iteration the current state is read. If, based on the current state, a transition is defined, the next state, symbol and direction for the tape head are retrieved from the transition table. The new symbol is written to the tape and the state is updated with the new state and symbol. The tape head is moved to the next position. If the position does not exist on the tape a blank is written. If the transition is not defined, the machine halts returning 0 if the state is accepted or 1 on reject. Finally lines 42 and 43 initialise the machine and line 45 starts execution.

Using the tables and function in Listings 7.3 and 7.4 any computable function can be programmed. Therefor, appropriate transition tables must be created which define the control flow to the function.

Hence according to, e.g. [3], SCSQL is Turing complete. ∎

7.5.2 Control Loop

A control loop is a universal pattern in industrial control systems. Control loops are comprised of controller, sensors and actuators. Their purpose is to regulate a variable set-point (SP), i.e. target condition under dynamic environment conditions. The widely applied Proportional-Integral-Derivative (PID) controllers try to minimise the error between the measured process value (PV) and the set-point by calculating appropriate actions, e.g. heat or cool. The proportional part of the controller constitutes a proportional response, $P_{out} = K_p e(t)$, to the error by multiplying the error $e = SP - PV$, with a constant K_p. The integral term $I_{out} = K_i * \int e(\tau) d\tau$, is proportional to the magnitude as well as

Listing 7.3: Turing Machine

```
1
2  STATE -> CREATE TABLE(STRING state,
3                        STRING symbol,
4                           INT pos)
5
6  TAPE -> CREATE TABLE(STRING symbol,
7                           INT pos)
8
9  TRANSITION -> CREATE TABLE(STRING state,
10                            STRING symbol,
11                               INT shift,
12                            STRING nextstate,
13                            STRING nextsymbol)
14
15 ACCEPT -> CREATE TABLE(STRING state)
```

duration of the error. Its contribution is controlled via the constant K_i. Eventually, the derivative part $D_{out} = K_d \frac{de}{dt}(t)$ contributes with the rate of change of the error over time multiplied with the constant K_d.

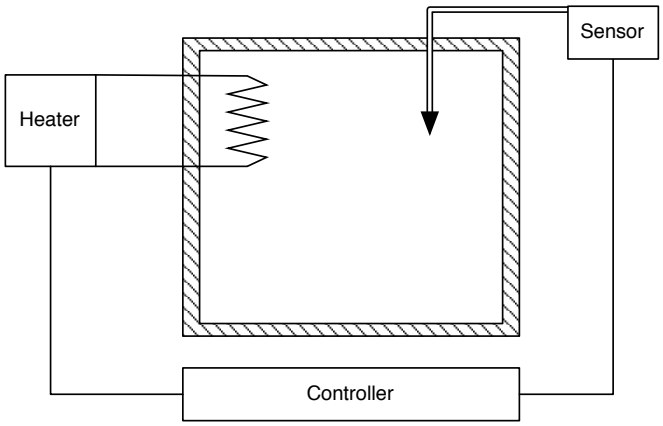


Figure 7.15: Simple control loop. Inspired by [87]

Control loops with PID controllers can be implemented with SCSQL compactly. To show that, consider the temperature control system depicted in Figure 7.15. It consists of a heater, temperature sensor and controller. As SCSQL abstracts form the underlying infrastructure the code to implement the control system is agnostic to the unit location. Each unit could be located on the same node or distributed over multiple nodes.

Listing 7.5 shows a SCSQL implementation of a PID controller. Lines 21-23 select the heaters to be controlled. They are specified as nodes of type “heater” and located in an area called “A1”. Lines 25-27 select corresponding sensors. Eventually, lines 29-33 initiate the control loop by passing process values from all sensors to the PID function which, in

turn, adjusts the temperature in the heater.

Apart from implementing a control loop, the example shows other characteristics of the architecture as well: declarative composition and de-normalised data. Firstly, the control system is specified at a high level of abstraction by describing parts by their attributes. This allows for the replacement of individual parts or system upgrades without the need to adjust the control algorithm. Concretely, the system could consist of one ore multiple heaters and sensors, sensors could be replaced or added. Independently of the change in the system, the controller will remain the same. The fact increases the level of flexibility and allows for systems that adapt to a changing infrastructure. Secondly, due to the data and table model data is de-normalised and hence the queries in lines 21 and 25 do not join nor use a foreign key relationship table. In contrast, in a strict relational model the system would be modelled with three tables, i.e. `heaters(id, ..)`, `sensors(id, ..)` and `temperaturesensors(heater id, sensor id)` or two tables, namely `sensors`, `heaters` and a join to select sensors that belong to the heaters of interest. If there were many heaters with each having many sensors, the `temperaturesensors` table would grow very large quickly. In the model introduced in this work, the relation between sensors and heaters is stored as column, hence it can be read efficiently, i.e. sequentially, which is the fastest way to retrieve the relationship even for many heater and sensor objects.

7.5.3 Complexity

The previous section introduced a simple control loop and identified it as a common pattern in industrial systems. In this section an analysis is conducted to determine the benefits of the query language when implementing control loops in distributed environments. The analysis is oriented towards the effort a developer must spent in order to achieve the functionality. Again, the complexity measure from Section 7.2 is applied to quantify efforts and complexity.

The development of distributed systems is particularly challenging due to the large variety of failures and exceptions that can occur. With increasing system size, the probability of *unusual* exceptions, e.g. network partitioning, oscillating interruptions or failure of equipment, increases proportionally. Developers of distributed programs must anticipate these behaviours and write additional code to compensate failures and handle exceptions.

Due to the high level of abstraction SCSQL provides, all communication related functionality, i.e. messaging, asynchronous calls etc., is hidden. Developers specify their interest in data declaratively and transparently of the actual storage location of the data. In the example code provided by Listing 7.5, data from a set of sensors is retrieved. This data

could be gathered directly from a sensor device which might be connected to a local machine. The sensor could also be installed in a remote location thousands of kilometres away or the sensor data might be read of a file or an archive database.

The declarative abstraction reduces the number of states a developer needs to take care of. According to Section 7.2.2, entropy is reduced if either the number of states decreases or the probability of a few states is significantly higher than that of most other states. Hence, with lower entropy the implementation complexity for the developer that uses SCSQL is reduced in comparision to a non data-centric approach. In the above example, state is kept only in the two tables `sensors` and `heaters`. On the contrary, the compiled version of the program handles nearly thousands of states to co-ordinate resource use, execute communication protocols and handle exceptions.

7.6 Summary

In this chapter, we evaluated the architecture described in Chapters 5 and 6. Achievement of quality attributes was shown by discussing the contributions of individual architectural patterns as well as reflection of applicability in the context of the scenarios. Using a quantitative complexity measure, critical parts of node and index cloud architecture were evaluated. The analysis showed quantitatively the fitness of the architectural concepts with regard to performance, availability, modifiability and adaptability. The index cloud is able to adjust to different workload patterns and hence can optimise its optimal throughput autonomously even under changing load conditions.

The cost based optimiser creates locally optimal query configurations given a set of optimisation criteria and information contained in local tables. On a global scale, however, it is not guaranteed to deliver fully optimal results. Yet already optimisation of simple queries has been proven to be complex (NP-hard) yielding considerable (not acceptable) effort. Hence, heuristic query optimisation methods are suggested to find query configurations close to the global optimum.

The network of query processors can exhibit complex behaviour. The strict scheduling kernel, the single threaded discrete processing model and the query optimiser contribute substantially to the stable operation of the network. The rejection behaviour of the kernel prevents node overload and system inefficiencies due to cascading effects. The option to optimise for entropy dampens global reallocations and achieves schedules closer to the global optimum. The node module concept, in particular the concentration of the complete node state in a single module, allows for advanced methods to automatically analyse health, resource utilisation, and processing efficiency of nodes.

The monitoring module provides the basis for query optimisation by ensuring the availability of up to date information of neighbouring node states. The Peer-to-Peer watchdog mechanism to detect node failures takes away considerable load from the index cloud. Monitoring module and query processor enable declarative clustering of nodes. Intensified communication of nodes within groups can be exploited to implement high availability service execution.

The implementation complexity of the query processor is low. The query processor including the optimiser have been implemented with 3500 lines of Java code. The clear code base provides a promising outlook for future industry grade commercial implementations on a variety of platforms. On the contrary to the closed processing system, the language based interface allows applications to modify access, pre-processing and information routes with evolving requirements.

The programming language defined in Chapter 6 is Turing complete and able to express any access and control problem in context of the described ecosystem for energy services. Besides its completeness, SCSQL allows to implement standard control problems compactly and intuitively.

The layered module architecture allows direct simulation of very large[11] networks of nodes. Using the distributed discrete event simulator of Chapter 4 only kernel and the communication module must be replaced by simulation counterparts. Key business logic like the query and program set remain exactly like in the actual deployment. Hence, simulations can check dynamics and correctness of a system with very high accuracy prior to the actual deployment.

[11] Millions.

Listing 7.4: Turing Machine Logic

```
1
2
3  FUNCTION TURING(STRING state, STRING symbol, INT pos) {
4
5      foreach(state, symbol, pos) {
6          M -> SELECT shift,nextstate,nextsymbol
7                 FROM TRANSITION
8                WHERE TRANSITION.state = state
9                  AND TRANSITION.symbol = symbol
10
11         IF (M) {
12             DELETE FROM TAPE
13              WHERE TAPE.pos = pos
14
15             INSERT INTO TAPE(M.nextsymbol, pos)
16         }
17         ELSE {
18             A -> SELECT *
19                    FROM ACCEPT
20                   WHERE ACCEPT.state = state
21
22             IF (A) {
23                 RETURN 1 /*accept*/
24             }
25             ELSE {
26                 RETURN 0 /*reject*/
27             }
28         }
29
30         NEXT -> SELECT symbol, pos
31                   FROM TAPE
32                  WHERE TAPE.pos = pos + M.shift
33
34         IF (NOT NEXT) {
35                 INSERT INTO TAPE('b', pos + M.shift)
36         }
37         ELSE {
38             INSERT INTO STATE(M.nextstate, M.nextsymbol, pos + M.shift)
39         }
40
41     }
42 }
43
44 INIT -> SELECT symbol, pos
45           FROM TABLE
46          WHERE pos=0;
47
48 INSERT INTO current('q0', INIT.symbol, 0)
49
50 SELECT TURNING(state, symbol, pos)
51   FROM CURRENT
52 WINDOW (0,10,1)
```

Listing 7.5: Control Loop

```
1  Derror -> 0
2
3  FUNCTION PID(Pv, Sp) {
4
5    Kp -> 100
6    Ki -> 0.9
7    Kd -> 1000
8
9    Error -> Sp - Pv
10   TotalError -> TotalError + Error
11
12   Pgain -> Kp * Error
13   Igain -> Ki * TotalError
14   Dgain -> Kd * (Error - Derror)
15   Derror -> Dgain
16
17   RETURN Pgain + Igain + Dgain
18 }
19
20
21 HEATERS -> SELECT * FROM NODES
22                    WHERE type = 'heater'
23                      AND location = 'A1'
24
25 SENSORS -> SELECT * FROM NODES
26                    WHERE connectedTo = 'heater'
27                      AND location = 'A1'
28
29 UPDATE HEATERS SET temperature=(
30                                 SELECT PID(temperature, 180)
31                                   FROM SENSORS
32                                  )
33                 WINDOW(0,FOREVER, 1)
```

Chapter 8

Conclusions and Future Work

Motivated by the requirements of current and near-future industrial systems, this work presented a generic data-centric architecture for large-scale industrial systems. The architecture itself implements the concept of the service ecosystem for energy services, i.e. an open platform for stakeholders in the Smart Grid domain to collaborate and conduct business. At the core of the ecosystem is the data which is generated, consumed, transferred and traded by stakeholders. The simplified data model, comprised of data items and tables, constitutes a lingua franca for all nodes connected by the ecosystem. The ecosystem itself builds upon three core services, namely (i) identification, (ii) registration and (iii) incentive to foster collaboration and engagement. By offering these three services, the ecosystem providers, the service providers and the service consumers can benefit.

The technical implementation and validation of the concept is the architecture presented in Chapter 6. It is based upon three key concepts, namely: (i) a modular node comprised of five core modules, (ii) an index cloud to provide global access to data and (iii) a data-centric query language, SCSQL, to organise, query and route data between nodes. The node architecture introduces a general computing paradigm. The five modules provide the required functionality to execute queries in a distributed context. By design, they also support assessment and stabilisation of large networks of query processors. Thereby the discrete execution model and the concentration of state in the memory module play a particularly important role.

The ability to incorporate several security mechanisms as well as the possibility to clearly identify the sender of a message, implements the identification service of the ecosystem. The index cloud, which aggregates meta-data of all nodes currently present in the ecosystem implements the registration service. The ability to concretely express quality requirements such as verifiability in combination with message signatures and security mechanisms provide the foundation for monetary incentive services.

The thorough evaluation conducted in Chapter 7 showed suitability and correctness of the architectural approach. Applied tactics and architecture evaluation methods of Chapter 4 showed the capabilities of the architecture to meet the quality requirements that where identified during the analysis of the scenarios in the beginning of Chapter 6. A prototypical implementation of nodes and index cloud allowed the quantitative evaluation of performance attributes. Expressiveness of the query language was proven such that broad applicability of the language to typical problems in the industrial domain is assured.

In Section 1.5, the research questions for this thesis were stated. Questions were structured in three categories namely: (i) Design and Evolution, (ii) Coordination and Control and (iii) Monitoring and Assessment. Chapters 5, 6 and 7 provided answers for each of these questions. The following summarises the contributions structured by the three categories.

8.1 Design and Evolution

The research questions in this category were: How can a system be designed that addresses all individual needs of its users and contributors? How can the system designed be evolved and adapted to changing policies and requirements? The answer has been elaborated in Chapters 5 and 6. The concept of the ecosystem for energy services provides the basic building blocks for all participants to join and extend the ecosystem. The minimalist data model constitutes the foundation for communication and co-ordination. The open character of the ecosystem allows for decentralised and collaborative engineering processes similar to the open source approach. The modular node architecture as well as the query processor concept and the query language enable agile modification and adoption of concrete technical systems.

8.2 Coordination and Control

Research questions of this category were: How can usage of shared resources be realised while maintaining the quality of service? How can the system be modified to adapt to new requirements or changes in the environment without considerable interruptions? How can users customise their interaction with other users? In the scenario analysis in Chapter 6 the set of requirements concretised these questions. The answer to these question was given by the modular node architecture as described in Section 6.3. The discrete runtime model of the kernel enables deterministic execution of queries. The strict scheduling policy as well as the query optimiser circumvent individual node overload. As shown by the backup protection scenario, modification or apposition of functionality, e.g. protection algorithms

or devices, is supported by the architecture either by writing corresponding queries in SCSQL or modification of the node modules. The high level of abstraction provided by SCSQL eases development efforts and reduces sources of failure.

8.3 Monitoring and Assessment

Due to the size of large-scale industrial systems, the system state can only be partially assessed. Hence the research questions in this category were: What are meaningful indicators that characterise the current system state? How can the potential effect of a control action be determined? Since complete state assessment is not possible and hence information is imprecise and uncertain, how do monitoring and assessment mechanisms cope with the constantly and quickly evolving states? Chapters 6 and 7 provided the answers. The node architecture subsumes the entire state of the node in a central module, namely the memory module. The module is accessible by the query language and hence can be included in query optimisation decisions. State evolution is discretised by the runtime model such that consistent state assessments also for small groups of nodes become possible. To estimate system behaviour, a measure for system complexity was introduced in Chapter 7. The query optimiser may use this information to make advanced scheduling decisions such that resource allocation overhead is reduced and distributed query execution stabilised.

8.4 Future Work

This work covers a particular section of ULS research by the example of power infrastructures. Surely a wide range of future work exists. While this work elaborated the key concepts and provided a prototypical implementation, a concrete deployment in an industrial setting has yet to be done. Promising candidates for a pilot installation constitute rural communes or isolated islands as their infrastructure is relatively small yet often based on modern technology.

Moreover further research has to be conducted to identify appropriate mechanisms to control entropy while operating the system close to its limits. In this context, the research field of networked control systems constitutes a promising starting point for further research. The methods described in Chapters 6 and 7 are rather strict, i.e. prevent relocation and suppress optimization. As a result optimal system utilisation cannot be achieved. In addition probilistic methods and machine learning may be applied to predict the effects on entropy for each query execution instruction.

Depending on the constraint set supplied with a query, executing nodes create new communication links for efficient and low latency communication. Chapters 6 and 7 supplied a gossip based clustering mechanism for nodes. In order to determine the semnatic proximity of nodes a simple average function was supplied. The correlation between measured values might be more complex and involve measurements of many different nodes. Hence, more advanced data mining methods, e.g. collective hiearchical clustering [73], might be needed. In this context the key challenge will be to ensure stability of the clustering algorithm in highly dynamic environment.

The data model and query language proposed in this thesis allow to describe and specify any object or process relevant in the ecosystem of energy services. Yet it does not define any domain specific semantics. To support, e.g., the seamless exchange of equipment semantic description of device capabilities might be beneficial. For example, slight variations in the measurement sensitivity of a temperature sensor might have substantial influcence on the stability of the system. Having a semantic description of a sensor allows for automatic compatibility checking and hence prevention of critical events. Additionally to the data model a domain specific set of core data objects should be developed. This base *ontology* will also ease the translation process from various standards to the lingua franca of the ecosystem.

In index cloud is capable to adapt to varying load patterns. To accomplish this, the master implements a control loop: measure the characteristic load for each index node, determine an optimal configuration of read and write zones, and finally, assign corresponding roles

to structure the cloud. In the current implementation the characteristic function of an index node is determined by running a series of test queries with a range of write/read loads. Throughput may also depend on other crieria, such as type and structure of the query, e.g conjuncts or type of data requested. Therefore research should be conducted to find an optimal set of probe queries and hence increase the accuracy of the characteristic function.

In context of this work a grammar, compiler, and runtime environment for SCSQL were implemented. Although the compiler provides basic support for failure detection, the current toolchain is not suitable for industrial grade applications. Further tools are required for development as well as debugging and test. Due to the high level of abstraction, the developer has only little influence on the final query configuration that is executed. For debugging purposes, the prototypical implementation of the toolchain allows for offline visualisation of query configurations via GraphViz [40]. However, particularly for dynamic re-configurations, real-time visualisation of the query execution is necessary. The eclipse platform might be a good starting point for a SCSQL development and test suite.

Chapters 5 and 6 focussed on power infrastructures. However, the concepts are generic and can be applied to a other domains as well. Flexible production systems that support modern product management strategies like mass-customization, contract manufacturing and modular automation constitute corresponding starting points.

Although Chapter 6 provided a prototypical solution for a role based access mechanism, security is not the primary focus of this work. The approach of Section 6.6.3 requires manual negotiation of attributes and signatures. In a scenario with thousands of partners, this manual process might not be appropriate. Future work should be spend on standardisation of security methods and access attributes in order to achieve fully automated, cross-entity processes.

Bibliography

[1] P2PSim A Simulator for Peer-to-Peer Protocols. http://pdos.csail.mit.edu/p2psim/.

[2] IEC 61000-4-30. *Electromagnetic compatibility (EMC) - Part 4-30: Testing and measurement techniques - Power quality measurement methods.* IEC, 2003.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[4] Jordi Pujol Ahulló and Pedro García López. Planetsim: an extensible framework for overlay network and services simulations. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–1, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[5] Lorenzo Alvisi, Jeroen Doumen, Rachid Guerraoui, Boris Koldehofe, Harry Li, Robbert Van Renesse, and Gilles Tredan. How robust are gossip-based communication protocols? *ACM SIGOPS Operating Systems Review*, 41(5):14–18, Oktober 2007.

[6] Mark Armstrong. Competition in two-sided markets. *RAND Journal of Economics*, 37:668–691, 2006.

[7] Rajive L. Bagrodia, K. Mani Chandy, and Jayadev Misra. A message-based approach to discrete-event simulation. *IEEE Trans. Softw. Eng.*, 13(6):654–665, 1987.

[8] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, New York, NY, USA, 2002. ACM.

[9] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete-Event System Simulation, Fourth Edition.* Prentice Hall, December 2004.

[10] Udo Bartlang and Jörg P. Müller. Dhtflex: A flexible approach to enable efficient atomic data management tailored for structured peer-to-peer overlays. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 377–384, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, Boston ; Munich [u.a.], 2003.

[12] Siegfried Behrendt, Florian Beissner, Daniel Doberstein, Lorenz Erdmann, Dr. Edgar Göll, Dr. Roland Nolte, Timon Wehnert, and Michaela Wölk. Integrierte Technologie-Roadmap Automation 2015+. Technical report, ZVEI - Zentralverband Elektrotechnik und Elektronikindustrie e.V., 2006.

[13] Daniel Bell. *The coming of post-industrial society: A venture in social forecasting*. Basic, New York, 1973.

[14] K. Bennett and V. Rajlich. Software evolution: A road map. *Software Maintenance, IEEE International Conference on*, 0:4, 2001.

[15] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: design, analysis and applications. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 3:1653–1664 vol. 3, 13-17 March 2005.

[16] Georg Bretthauer, Gerald Gerlach, Friedrich Harbach, and Dieter Westerkamp. Automation 2020. Technical report, VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik (GMA), 2009.

[17] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[18] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.

[19] Stefan Bussmann, Nicolas R. Jennings, and Michael Wooldridge. *Multiagent Systems for Manufacturing Control*. SpringerVerlag, 2004.

[20] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[21] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[22] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[23] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, 1979.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[25] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[26] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, January 2002.

[27] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[28] CRUTIAL. Analysis of new control applications. Deliverable D2, 2006. Critical Utility Infrastructural Resilience, EU Project IST-FP6-STREP-027513.

[29] T. Cucinotta, A. Mancina, G.F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina. A real-time service-oriented architecture for industrial automation. *Industrial Informatics, IEEE Transactions on*, 5(3):267–277, Aug. 2009.

[30] C. J. Date. A critique of the SQL database language. *SIGMOD Rec.*, 14(3):8–54, 1984.

[31] Ian Davis and Elizabeth Stephenson. Ten trends to watch in 2006. *The McKinsey Quarterly*, (1), 2006.

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[33] Ewa Deelman, Rajive Bagrodia, Rizos Sakellariou, and Vikram Adve. Improving lookahead in parallel discrete event simulations of large-scale applications using compiler analysis. In *PADS '01: Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pages 5–13, Washington, DC, USA, 2001. IEEE Computer Society.

[34] S. E. Deering. Multicast routing in internetworks and extended lans. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 55–64, New York, NY, USA, 1988. ACM.

[35] Juancarlo Depablos. Internet peer-to-peer communication based distribution loop control system. Master's thesis, Virginia Tech Polytechnic Institute, 2003.

[36] Hrishikesh Deshpande, Mayank Bawa, and Hector Garcia-Molina. Streaming live media over a peer-to-peer network. August 2002.

[37] C. Diot, B.N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the ip multicast service and architecture. *Network, IEEE*, 14(1):78–88, Jan/Feb 2000.

[38] Kolja Eger, Christoph Gerdes, and Sebnem Öztunali. Towards P2P technologies for the control of electrical power systems. In *P2P '08: Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, pages 180–181, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Thomas Eisenmann, Geoffrey Parker, and Marshall W. Van Alstyne. Strategies for two-sided markets. *Harvard Business Review*, 84:92–101, 2006.

[40] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. *Graph Drawing*, pages 483–484, 2001.

[41] European Technology Platform Smart Grid. Strategic deployment document. Technical report, European Commission, 2008.

[42] European Technology Platform Smart Grid. Strategic research agenda for europes electricity networks of the future. Technical report, European Commission, 2008.

[43] Eckhard D. Falkenberg, Wolfgang Hesse, Paul Lindgreen, Bjrn E. Nilsson, J. L. Han Oei, Colette Rolland, Ronald K. Stamper, Frans J. M. Van Assche, Alexander A. Verrijn-stuart, and Klaus Voss. The frisco report (web edition), 1996.

[44] Marc Fleury and Francisco Reverbel. The jboss extensible server. pages 344–373. Springer, 2003.

[45] Force.com. Salesforce object query language (soql). `http://wiki.developerforce.com/index.php/BNF_FOR_SOQL`.

[46] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, 2005.

[47] International Monetary Fund. World economic outlook : a survey by the staff of the international monetary fund. *World economic outlook : a survey by the staff of the International Monetary Fund*, pages 110–120, 2002.

[48] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams (Data-Centric Systems and Applications)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[49] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, 2004.

[50] Christoph Gerdes, Udo Bartlang, and Jörg P. Müller. Decentralised and reliable service infrastructure to enable corporate cloud computing. In Paul Cunnigham and Miriam Cunnigham, editors, *Collaboration and the Knowledge Economy: Issues, Applications and Case Studies*, volume 5 of *Information and Communication Technologies and the Knowledge Economy*, pages 683–690, Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands, October 2008. IIM, IOS Press. Proceedings of eChallenges e-2008 Conference.

[51] Christoph Gerdes, Kolja Eger, and Jörg Müller. Data centric peer-to-peer communication in power grids. *ECEASST*, 17, 2009.

[52] Christoph Gerdes, Christian Kleegrewe, and Jörg P. Müller. Declarative resource discovery in distributed automation systems. In I. Troch and F. Breitenecker, editors, *MathMod*, volume ARGESIM Report, 2009.

[53] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[54] Lukasz Golab and M. Tamer Oezsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[55] Google. GQL reference. `http://code.google.com/intl/de-DE/appengine/docs/python/datastore/gqlreference.html`.

[56] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, Marseille, France, November 2002.

[57] A. Gunasekaran and E.W.T. Ngai. Build-to-order supply chain management: a literature review and framework for development. *Journal of Operations Management*, 23(5):423 – 451, 2005. The Build to Order Supply Chain (BOSC).

[58] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 242–259, London, UK, 2002. Springer-Verlag.

[59] N.D. Hatziargyriou, A. Dimeas, A.G. Tsikalakis, J.A. Pecas Lopes, G. Kariniotakis, and J. Oyarzabal. Management of microgrids in market environment. *Future Power Systems, 2005 International Conference on*, pages 1–7, 16-18 Nov. 2005.

[60] M.J. Hawthorne and D.E. Perry. Architectural styles for adaptable self-healing dependable systems. In *In Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE 2005)*, 2005.

[61] Joseph M. Hellerstein. Programming a parallel future. Technical Report UCB/EECS-2008-144, EECS Department, University of California, Berkeley, Nov 2008.

[62] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a Database System.* Now Publishers Inc., Hanover, MA, USA, 2007.

[63] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Embedded control systems development with giotto. In *OM '01: Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, pages 64–72, New York, NY, USA, 2001. ACM.

[64] Katja Hose, Christian Lemke, Jana Quasebarth, and Kai-Uwe Sattler. Smurfpdms: A platform for query processing in large-scale pdms. In Alfons Kemper, Harald Schning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *BTW*, volume 103 of *LNI*, pages 621–624. GI, 2007.

[65] R. Huebsch, B. Chun, J. Hellerstein, B. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. Yumerefendi. The architecture of pier: an internet-scale query processor, 2005.

[66] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *vldb'2003: Proceedings of the 29th international conference on Very large data bases*, pages 321–332. VLDB Endowment, 2003.

[67] Carolyn J. Hursch and Jack L. Hursch. *SQL: the structured query language.* TAB Books, Blue Ridge Summit, PA, USA, 1988.

[68] IEC/ISO. *IEC61850 Part 7-3: Basic Communication Structure for Substation and Feeder Equipment - Common Data Classes.* IEC, Geneva, Switzerland, 2004.

[69] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* IEEE, New York, August 12 1985.

[70] A. Ipakchi and F. Albuyeh. Grid of the future. *Power and Energy Magazine, IEEE*, 7(2):52–62, March-April 2009.

[71] Ali Ipakchi. Implementing the smart grid: Enterprise information integration. *GridInterop Forum 2007*, 2007.

[72] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[73] Erik L. Johnson and Hillol Kargupta. Collective, hierarchical clustering from distributed, heterogeneous data. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 221–244, London, UK, 2000. Springer-Verlag.

[74] M. Kim, M.J. Damborg, J. Huang, and S.S. Venkata. Wide-area protection using distributed control and high speed communications. In *14th Power Systems Computation Conference*, 2002.

[75] Hermann Kopetz. *Real-Time Systems-Design Principles for Distributed Embedded Applications.* Kluwer Academic publishers, 1997.

[76] Jouni Korhonen. Four ecosystem principles for an industrial ecosystem. *Journal of Cleaner Production*, 9(3):253 – 259, 2001.

[77] Niko Kotilainen, Mikko Vapa, Teemu Keltanen, Annemari Auvinen, and Jarkko Vuori. P2prealm peer-to-peer network simulator, in. In *Proc. 11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks, 2006*, pages 93–99. Unpublished, 2006.

[78] Stefan Saroiu Krishna P. Gummadi and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. *Proceedings of SIGCOMM IMW 2002*, 2002.

[79] Bhaskar Krishnamachari, Deborah Estrin, and Stephen B. Wicker. The impact of data aggregation in wireless sensor networks. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 575–578, Washington, DC, USA, 2002. IEEE Computer Society.

[80] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[81] D. Kucuk, B. Boyrazoglu, and S. Buhan. Pqstream: A data stream architecture for electrical power quality. In *International Workshop on Knowledge Discovery from Ubiquitous Data Streams*, 2007.

[82] Minseok Kwon and Sonia Fahmy. Topology-aware overlay networks for group communication. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 127–136, New York, NY, USA, 2002. ACM.

[83] Leslie Lamport. The mutual exclusion problem: part i - a theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986.

[84] R. Lasseter, A. Akhil., C. Mamay, J. Stephens, J. Dagle, R. Gullromson, A. Meliopoulos, R. Yinger, and J. Eto. White paper on integration of distributed energy resources - the certs microgrid concept. *California Energy Comission*, 2002.

[85] R.H. Lasseter. Microgrids. *Power Engineering Society Winter Meeting, 2002. IEEE*, 1:305–308 vol.1, 2002.

[86] Jan Lemeire, Wouter Brissinck, and Erik Dirkx. 1 lookahead accumulation in conservative parallel discrete event simulation. In *Proceedings 18th European Simulation Multiconference*, 2008.

[87] R. W. Lewis. *Modelling Distributed Control Systems Using IEC 61499*. Institution of Electrical Engineers, Stevenage, UK, UK, 2001.

[88] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[89] Jie Liu and E.A. Lee. Timed multitasking for real-time embedded software. *Control Systems Magazine, IEEE*, 23(1):65–75, Feb 2003.

[90] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 2006. ACM.

[91] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 75–90, New York, NY, USA, 2005. ACM.

[92] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. *SIGCOMM Comput. Commun. Rev.*, 35(4):289–300, 2005.

[93] J.A.P. Lopes, C.L. Moreira, and A.G. Madureira. Defining control strategies for microgrids islanded operation. *Power Systems, IEEE Transactions on*, 21(2):916–924, May 2006.

[94] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[95] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[96] Jörg P. Müller. *The Design of Intelligent Agents: A Layered Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[97] P. Naur and B. Randell. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, 1969.

[98] Linda Northrop, Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, 2006.

[99] The Network Simulator ns-2 (v2.1b8a). http://www.isi.edu/nsnam/ns/, October 2001.

[100] Open Application Integration Specification (OAGIS). *Open Application Integration Specification (OAGIS)*. Open Application Group, 2008.

[101] M. Tamer Oezsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[102] F. Van Overbeeke and V. Roberts. Active networks as facilitators for embedded generation. *Cogeneration and on-site power production,*, 3, 2002.

[103] Manish Parashar. *Autonomic Computing: Concepts, Infrastructure, and Applications*. Taylor & Francis, Inc., Bristol, PA, USA, 2007.

[104] Geoffrey Parker and Marshall W. Van Alstyne. Information complements, substitutes, and strategic product design. In *ICIS '00: Proceedings of the twenty first international conference on Information systems*, pages 13–15, Atlanta, GA, USA, 2000. Association for Information Systems.

[105] H. Van Dyke Parunak. *Multiagent systems: a modern approach to distributed artificial intelligence*, chapter Industrial and practical applications of DAI, pages 377–421. MIT Press, Cambridge, MA, USA, 1999.

[106] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. Almi: an application level multicast infrastructure. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 5–5, Berkeley, CA, USA, 2001. USENIX Association.

[107] P. Piagi and R.H. Lasseter. Autonomous control of microgrids. *Power Engineering Society General Meeting, 2006. IEEE*, pages 8 pp.–, 18-22 June 2006.

[108] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[109] Joseph B. Pine. *Mass customization: the new frontier in business competition*. Harvard Business School, Boston, Mass., 1993.

[110] Vivian Prinz, Florian Fuchs, Peter Ruppel, Christoph Gerdes, and Alan Southall. Adaptive and fault-tolerant service composition in peer-to-peer systems. In *DAIS*, pages 30–43, 2008.

[111] James D. Proctor and Brendon M. Larson. Ecology, complexity, and metaphor. *BioScience*, 55(12):1065–1068, December 2005.

[112] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[113] Ananth Ranganathan. The levenberg-marquardt algorithm, 2004.

[114] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content addressable network. Technical Report TR-00-010, Intel Research Berkeley, New York, NY, USA, 2001.

[115] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[116] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., 2001.

[117] Christian Rehtanz. *Autonomous Systems and Intelligent Agents in Power System Control and Operation (Power Systems)*. SpringerVerlag, 2003.

[118] M. Ripeanu. Peer-to-peer architecture case study: Gnutella networks. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing*, page 99, Washington, DC, USA, 2001. IEEE Computer Society.

[119] Daniel. Robey. *Designing organizations*. Irwin, Homewood, IL, 3rd ed. edition, 1991.

[120] Vincent Roca and Ayman El-Sayed. A Host-Based Multicast (HBM) Solution for Group Communications. In *ICN '01: Proceedings of the First International Conference on Networking-Part 1*, pages 610–619, London, UK, 2001. Springer-Verlag.

[121] Jean C. Rochet and Jean Tirole. Two-sided markets: A progress report. *RAND Journal of Economics*, 37:645–667, 2006.

[122] Jean-Charles Rochet and Jean Tirole. Platform competition in two-sided markets. *Journal of the European Economic Association*, 1(4):990–1029, 06 2003.

[123] Stephan Roser. *Designing and Enacting Cross-organisational Business processes: A Model-driven, Ontology-based Approach*. PhD thesis, Department of Computer Science University of Augsburg, 2008.

[124] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[125] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC '01:*

Proceedings of the Third International COST264 Workshop on Networked Group Communication, pages 30–43, London, UK, 2001. Springer-Verlag.

[126] Pieter Schavemaker and Lou van der Sluis. *Electrical Power System Essentials*. John Wiley & Sons Ltd, 2008.

[127] S. Schoenherr. Wireless technologies for distribution automation. *Transmission and Distribution Conference and Exposition, 2003 IEEE PES*, 1:375–378 Vol.1, Sept. 2003.

[128] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *AUUG97*, 1997.

[129] Srinarayan Sharma, Vijayan Sugumaran, and Balaji Rajagopalan. A framework for creating hybrid-open source software communities. *Inf. Syst. J.*, 12(1):7–26, 2002.

[130] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. Overlay weaver: An overlay construction toolkit. *Comput. Commun.*, 31(2):402–412, 2008.

[131] Atul Singh, Petros Maniatis, Timothy Roscoe, Timothy Roscoe, and Peter Druschel. Using queries for distributed monitoring and forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, 2006.

[132] Adam Smith and Alan B. Krueger. *The Wealth of Nations*. Bantam Classics, 2003.

[133] Diomidis Spinellis. Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99, 2001.

[134] F. Stäber, C. Gerdes, and J.P. Müller. A peer-to-peer-based service infrastructure for distributed power generation. In *Proc. of 17th IFAC World Congress, Seoul, Korea, Intl.l Federation of Automatic Control*, 2008.

[135] Herbert. Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien ; New York, 1973.

[136] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[137] D.B. Stewart, R.A. Volpe, and P.K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *Software Engineering, IEEE Transactions on*, 23(12):759–776, Dec 1997.

[138] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications.

In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[139] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal: Very Large Data Bases*, 5(1):48–63, 1996.

[140] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998.

[141] D. Stutzbach and R. Rejaie. Towards a better understanding of churn in peer-to-peer networks. Technical report, Department of computer science, University of Oregon, November 2004.

[142] Hbase Development Team. Hbase: Bigtable-like structured storage for hadoop hdfs, 2007.

[143] Nyik San Ting and Ralph Deters. 3ls a peer-to-peer network simulator. *Peer-to-Peer Computing, IEEE International Conference on*, 0:212, 2003.

[144] D.A. Tran, K.A. Hua, and T.T. Do. A peer-to-peer architecture for media streaming. *Selected Areas in Communications, IEEE Journal on*, 22(1):121–133, Jan. 2004.

[145] Richard Veryard. *Component-based business: plug and play.* Springer, 2000.

[146] Birgit Vogel-Heuser, Gunther Kegel, Klaus Bender, and Klaus Wucherer. Global infomation architecture for industrial automation. *Automatisierungstechnische Praxis*, 51:108–115, 2009.

[147] A. Vojdani. Smart integration. *Power and Energy Magazine, IEEE*, 6(6):71–79, November-December 2008.

[148] Leonard Waverman and Esen Sirel. European telecommunications markets on the verge of full liberalization. *The Journal of Economic Perspectives*, 11(4):113–126, 1997.

[149] Yuan Wei, Vibha Prasad, and Sang H. Son. Qos management of real-time data stream queries in distributed environments. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 241–248, Washington, DC, USA, 2007. IEEE Computer Society.

[150] Yuan Wei, Sang H. Son, and John A. Stankovic. Rtstream: Real-time query processing for data streams. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[151] Gio Wiederhold. *Database Design.* McGraw-Hill Book Company, 1977.

[152] Wikipedia. Energiewirtschaftsgesetz — wikipedia, die freie enzyklopdie, 2009. [Online; Stand 18. Februar 2010].

[153] Wikipedia. Finanzsystem — wikipedia, die freie enzyklopdie, 2009. [Online; Stand 18. Februar 2010].

[154] Wikipedia. System — wikipedia, the free encyclopedia, 2009. [Online; accessed 2-November-2009].

[155] Wikipedia. Informationsarchitektur — wikipedia, die freie enzyklopdie, 2010. [Online; Stand 21. April 2010].

[156] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems*, pages 68–77, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[157] W. Yang and N. Abu-Ghazaleh. Gps: a general peer-to-peer simulator and its use for modeling bittorrent. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 425–432, Sept. 2005.

[158] Franco Zambonelli and Mirko Viroli. Architecture and metaphors for eternally adaptive service ecosystems. *Intelligent Distributed Computing, Systems and Applications*, 162/2008:23–32, 2008.

[159] Rongmei Zhang and Y. Charlie Hu. Borg: a hybrid protocol for scalable application-level multicast in peer-to-peer networks. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 172–179, New York, NY, USA, 2003. ACM.

[160] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[161] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM.

Appendices

Appendix A

Cost Functions

$$cost_{latency}(Q) = max_i(cost_{responsetime}(q_i)) \tag{A.1}$$

with q_i all sub-queries required to process Q.

$$cost_{responsetime}(Q) = t_a - t_s \tag{A.2}$$

with t_a the query execution starts and t_s the time, the first result arrived.

$$cost_{throughput}(Q) = w_{throughput} \times max(1, e^{\frac{t_s - t_m}{t_s}}) \tag{A.3}$$

with t_s the throughput specified and t_a throughput measured.

$$cost_{reliability}(Q) = w_{throughput} \times e^{\frac{t_s - t_m}{t_s}} \tag{A.4}$$

with t_s MTBF specified and t_m MTBF measured.

$$cost_{trust}(Q) = w_{trust} \times \delta(trustlevel) \tag{A.5}$$

where $\delta(x)$ is 1 iff trustlevel = local trust level and ∞ otherwise.

Appendix B

SCSQL Grammar

```
statement
    :
    (functionDef)*
    (selectStatement)+
    EOF
    -> ^(PROGRAM (functionDef)* (selectStatement)+ )
    ;

functionDef
    :
    FUNCTION
    functionType
    Identifier
    functionHead
    functionBody
    -> ^(FUNCTION ^(FUNCTION_NAME Identifier)
                                functionHead
                                functionBody ^(FUNCTION_RETURN functionType))
    ;

functionType
    :
    INT | FLOAT | STRING
    ;

functionHead
    :
    '('
    Identifier
    (COMMA Identifier)*
    ')' -> ^(FUNCTION_PARAMETERS (Identifier)+)
    ;
```

```
functionBody
    :
    LCURLY
    (variableTypeDef)*
    (subStatement)+
    RETURN Identifier SEMI
    RCURLY  -> ^(FUNCTION_BODY (variableTypeDef)* (subStatement)+ ^(RETURN Identifier) )
    ;

variableTypeDef
    :
    INT Identifier SEMI     -> ^(INTEGER_DEF Identifier)
    | FLOAT Identifier SEMI     -> ^(FLOAT_DEF Identifier)
    ;

subStatement
    :
    assignmentExpr      -> ^(assignmentExpr)
    | forEachStatement  -> ^(forEachStatement)
    | ifStatement       -> ^(ifStatement)
    ;

assignmentExpr
    :
    Identifier '=' conditionalExpression SEMI
                                    -> ^(ASSIGNMENT Identifier conditionalExpression)
    ;

forEachStatement
    :
    FOREACH
    Identifier
    forEachBody -> ^(FOREACH Identifier forEachBody)
    ;

forEachBody
    :
    LCURLY
    (subStatement)+
    RCURLY  -> ^(FOREACH_BODY (subStatement)+)
    ;

ifStatement
    :
    IF
    LPAREN
    conditionalExpression
    RPAREN
    ifBody -> ^(IF conditionalExpression ifBody)
    ;
```

```
ifBody
    :
    LCURLY
    (subStatement)+
    RCURLY  -> ^(IF_BODY (subStatement)+)
    ;

conditionalExpression
    :
    orConditionalExpression
    ;

orConditionalExpression
    :
    andConditionalExpression
    ('||'^ andConditionalExpression)*
    ;

andConditionalExpression
    :
    comparisonConditionalExpression
    ('&&'^ comparisonConditionalExpression)*
    ;

comparisonConditionalExpression
    :
    arithmeticExpression
    ((EQUALTO^
     | GREATERTHANOREQUALTO^
     | LESSTHANOREQUALTO^
     | NOTEQUALTO^
     | GREATERTHAN^
     | LESSTHAN^)
    arithmeticExpression)*
    ;

arithmeticExpression
    :
    additiveExpression
    (bitwiseOperator^ additiveExpression)*
    ;

additiveExpression
    :
    multDivExpression (('+'^ | '-'^) multDivExpression)*
    ;

multDivExpression
    :
    expressionAtom ((STAR^ | DIVIDE^ | MOD^) expressionAtom)*
    ;
```

```
expressionAtom
    :
    Number
    | Identifier
    | LPAREN! conditionalExpression RPAREN!
    ;

selectStatement
    :
    (assignment)?
    selectClause
    fromClause
    (whereClause)?
    -> ^(SELECT_STATEMENT (assignment)? selectClause fromClause (whereClause)? )
    ;

assignment
    :
    Identifier
    ASSIGN
    -> ^(ASSIGNMENT Identifier)
    ;

selectClause
    : SELECT selectList -> selectList
    ;

selectList options { k=2; }
    :
    selectItem  (COMMA selectItem)*
        -> ^(WHAT_CLAUSE (selectItem)+ )
    ;

selectItem  options {k=2;}
    :
    selectExpression (AS Identifier)?
                      -> ^(SELECT_ITEM selectExpression (^(AS Identifier))? )
    ;


selectExpression  options {k=2;}
    :
    tableColumn  -> ^(TABLE_COLUMN tableColumn)
    | functionReference
    ;
    catch [NoViableAltException e]
    {
        e.grammarDecisionDescription = "Error parsing select item: ";
        throw e;
    }
```

```
tableColumn options {k = 2;}
    :
    Identifier
    | Identifier DOT Identifier -> ^(Identifier Identifier)
    ;

functionReference
    :
    Identifier
    (LPAREN
    (
    tableColumn
    (COMMA tableColumn)*
    )
    RPAREN)
        -> ^(FUNCTION ^(FUNCTION_NAME Identifier)
        ^(FUNCTION_PARAMETERS (tableColumn)+ )
        )
    ;


fromClause
    :
    FROM tableSource
    (COMMA tableSource)*
    (WINDOW LPAREN windowSpecification RPAREN)?
    (RECEIVER EQUAL receiverSpecification)?
        ->^(FROM_CLAUSE tableSource (tableSource)*
                        (windowSpecification)?
                        (receiverSpecification)?
            )
    ;

tableSource
    :
    Identifier
    ;
    catch [NoViableAltException e]
    {
        e.grammarDecisionDescription = "Error parsing table source: ";
        throw e;
    }

windowSpecification
    :
    Number SEMI Number SEMI Number -> ^(WINDOW_SPECIFICATION Number Number Number)
    ;

receiverSpecification
    :
    Identifier  -> ^(RECEIVER_SPECIFICATION Identifier)
    ;
```

```
whereClause
    :
    WHERE whereConditional -> ^(WHERE_CLAUSE whereConditional)
    ;

whereConditional
    :
    whereAndCondition
    (OR^ whereAndCondition)*
    ;

whereAndCondition
    :
    whereSubCondition
    (AND^ whereSubCondition)*
    ;

whereSubCondition
        :
        (NOT )?
            ( (LPAREN whereConditional RPAREN) => LPAREN! whereConditional RPAREN!
        | wherePredicate
    )
    ;

wherePredicate
        :
        whereExpression (
        comparisonOperator^ whereExpression
        | 'like' StringLiteral
        | 'in' LPAREN (constantSequence) RPAREN
    )
    ;

whereExpression
    :
      whereMultExpression
      ((PLUS^ | MINUS^) whereMultExpression)*
    ;

whereMultExpression
    :
    whereBitwiseExpression
    ((STAR^ | DIVIDE^ | MOD^) whereBitwiseExpression)*
    ;

whereBitwiseExpression
    :
    whereAtomExpression
    (bitwiseOperator^ whereAtomExpression)*
    ;
```

```
whereAtomExpression
        :
        (unaryOperator )?
        (
        constant
        | LPAREN! whereExpression RPAREN!
        | tableColumn
        )
        ;

constantSequence
    :
    constant
    (COMMA constant )*
;

constant
    :
    Number
    | StringLiteral
    | booleanValue
    ;

unaryOperator
    :
    MINUS | TILDE
    ;

    binaryOperator
    :
    arithmeticOperator | bitwiseOperator
    ;

arithmeticOperator
    :
    PLUS | MINUS | STAR | DIVIDE | MOD
    ;

bitwiseOperator
    :
    AMPERSAND | TILDE | BITWISEOR | BITWISEXOR
    ;

comparisonOperator
    :
    EQUAL | NOTEQUAL | LESSTHANOREQUALTO
    | LESSTHAN | GREATERTHANOREQUALTO |  GREATERTHAN
    ;

logicalOperator
    :
    'all' | 'and' | 'any' | 'exists' | 'in' | 'like' | 'not' | 'or' | 'some'
    ;
```

```
booleanValue
    :
    'true' | 'false'
    ;

DOT : '.';
COLON : ':' ;
COMMA : ',' ;
SEMI : ';' ;

LPAREN : '(' ;
RPAREN : ')' ;
LSQUARE : '[' ;
RSQUARE : ']' ;
LCURLY : '{' ;
RCURLY : '}' ;

EQUAL : '=' ;
EQUALTO :    '==' ;
NOTEQUAL : '<>' ;
NOTEQUALTO : '!=' ;
LESSTHANOREQUALTO : '<=' ;
LESSTHAN : '<' ;
GREATERTHANOREQUALTO : '>=' ;
GREATERTHAN : '>' ;



fragment
Letter
    : 'a'..'z' | '_' | 'A'..'Z'
    ;

fragment
Digit
    :
    '0'..'9'
    ;

StringLiteral
    :
    '\'' (~'\'')* '\'' ( '\'' (~'\'')* '\'' )*
    ;
Number
    :
    (Digit)+ ( DOT (Digit)*)?
    ;

Identifier
    :
    Letter (Letter | Digit)*
    ;
```

```
WS
      :
      (' '|'\r'|'\t'|'\n' | '\r\n') {skip();}
;

ASSIGN
      :
      '->'
      ;
```

Appendix C

Value Networks

Two-sided networks also referred to as *two-sided markets* are economic networks that link two distinct user groups in order to engage in some form of transaction. Two-sided markets can be found in various industries. The most well known example is the credit card which connects consumers with merchants. Other examples include Web 2.0 platforms, search engines and newspapers which link, e.g. advertises and readers or users. In general two-sided networks can be found in both product and service industries. The concept that implements a two-sided network is called a *platform*. It provides an infrastructure and basic services to facilitate transactions between the user groups. The platform aims to attract each user group and charges for its core services.

In contrast to traditional value chains where value is generated from left to right, in two-sided networks cost and revenue flows in both directions (Figure C.1). Hence value is generated from the interaction of the two user groups. The platform, acts as mediator and often takes a share off each transaction. Both user groups may be charged equally or the usage of one user group may be subsidised, e.g. the merchant pays a fee for each

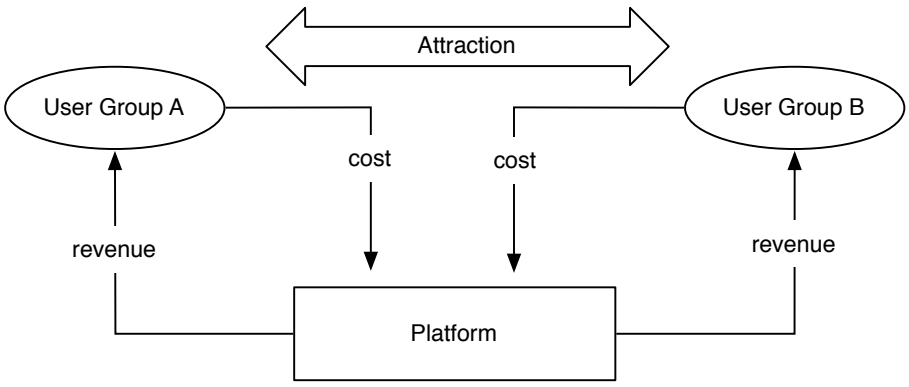


Figure C.1: Two-sided markets. Flow of revenue and cost. Value creation.

transaction, while for the buyer the transaction is free of charge. Platforms may also raise a membership fee which is neither trade nor usage based.

The phenomenon that the platform leverages, i.e. the attraction of both user groups to each other is called *network effect*. The value of the platform for a user is a function of the size of the user base on the other side of the network. Value increases when the platform matches demand from both sides. For example, consider the Microsoft operating system: the more people use the operating system, the more attractive it is for software developers to offer their products on the Microsoft platform. On the other hand, a large variety of software products increases the attractiveness for the platform user. Other popular examples are game platforms such as Atari, Nintendo, Play Station, Microsoft X-Box which try to attract both gamers and game developers.

Supported by the network effects, successful platforms achieve higher increases in returns of scale. Due to the increased attractiveness, users will pay more for access to a bigger network. In turn this yields higher margins with growing numbers of users. This behaviour is in contrast to traditional businesses where growth of the user base leads to decreased margins [39].

Cultural change and technological advancements such as the Internet have boosted the importance of platforms drastically. Popular platforms such as the Google search engine, link millions of advertisers and web searchers world wide and generate multi billion Euro volumes. However, two-sided markets also appear in other industries such as manufacturing and utilities. With liberalised power infrastructures and the challenges inherent to integration of renewable sources, electricity markets migrate towards platforms that link electricity consumers and producers. Rochet and Tirole [122] provide further examples of successful two-sided markets.

In recent years research on two-sided markets has been intense. An overview of the field is given by Rochet et. al. [121]. A considerable body of research has been conducted on corresponding strategies e.g. Eisenmann [39], Armstrong [6] and Rochet [122] as well as Parker [104]. Platforms where more than two distinct user groups participate, implement *multi-sided markets*. The theory for two-sided markets can be generalised to multi-sided markets [6].

A prerequisite for successful implementation of a two-sided market is to leverage the network effects. This can be achieved by choosing an appropriate incentive scheme, e.g. pricing (embership fee or usage based). Moreover, the right balance between subsidisation and charging must be tuned to the target user groups [39].

Appendix D

Acronyms

ACID Atomicity, Consistency, Integrity and Durability

AGC Automatic Generation Control

ALM Application Layer Multicast

CMB Chandry Misra Bryant (protocol)

DER Distributed Energy Resource

DHT Distributed Hash Table

DS Distributed Storage (energy)

DSL Domain Specific Language

DSMS Data Stream Management System

EBNF Enhanced Backus-Naur Form

EDF Earliest Deadline First

EEG Erneuerbare Energien Gesetz

EMS Energy Management System

ERP Enterprise Resource Planning

GFS Google File System

HMI Human Machine Interface

HV High Voltage

IC Information and Communication

ICT Information and Communication Technology

IDE Integrated Development Environment

IP Internet Protocol

ISP Internet Service Provider

LFC Load Frequency Control

LV Low Voltage

MES Manufacturing Execution System

MST Minimum Spanning Tree

OSI Open System Interconnection Reference Model

OSM Open Source Model

P2P Peer-to-Peer

PLC Programmable Logic Controller

PQ Power Quality

RDBMS Relational Database Management System

RM Rate Monotonic

RTT Round Trip Time

RTU Remote Terminal Unit

SCADA Supervisory Control And Data Acquisition

SCSQL Service eCoSystem Query Language

SQL Structured Query Language

T&D Transmission and Distribution

TM Time Mutlitasking

ULS Ultra Large Scale Systems

WAC Wide Area Control

WAM Wide Area Monitoring

WCET Worst Case Execution Time

Curriculum Vitae

Christoph Gerdes Innere Wiener Str. 24 81667 München +49 179 9485406 c.gerdes@gmail.com

Ausbildung

08.1988–06.1997	Ludwig-Georgs-Gymnasium, Darmstadt (Humanistisches Gymnasium)
Juni 1997	Abitur
08.1997–09.1998	Zivildienst Deutsches Rotes Kreuz Rettungsdienst
10.1998–06.2004	Studium des Informatik-Ingenieurwesens, Technische Universität Hamburg Harburg. Schwerpunkte im Hauptstudium: Software Systeme, Verteilte Systeme, Ubiquitous Computing. Thema der Diplomarbeit: Aggregation in Interconnected Sensor Networks
8.2002–9.2003	Internationaler Master Studiengang Complex Adaptive Systems, Technische Universität Chalmers in Göteborg Schweden. Schwerpunkte: Simulation von komplexen Systemen, Künstliche Intelligenz, evolutionäre Methoden.
10.2003	Master Of Science in Engineering mit Auszeichnung
06.2004	Dipl. -Ing., Informatik-Ingenieurwesen
04.2005	Beginn der Promotion

Siemens AG

10.2004–09.2006	Research Scientist, Siemens AG, Corporate Technology
10.2006–09.2008	Projektleiter, Siemens AG, Corporate Technology
10.2008–heute	Program Manager, Siemens AG, Corporate Technology

---

Stipendien

Erasmus Auslandsstipendium

Ditze Stipendium

Adlerbertska Hospitiefonden Scholarship

Siemens Junior Top Talent

Besondere Kentnisse

Sprachen	Englisch fließend in Schrift und Sprache (TOEFL-Score 270 im Sommer 2002)
	Schwedischkenntnisse
	Französisch Grundkenntnisse

München, 31. Mai 2010