CONTEXT-AWARE SERVICE REGISTRY: MODELING AND IMPLEMENTATION

Alaa Alsaig

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

November 2013

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Miss. Alaa AbdulBasit Alsaig**

Entitled: **Context-Aware Service Registry: Modeling and Implementation**

and submitted in partial fulfillment of the requirements for the degree of

Master in Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. N. Tsantalis

_____ Examiner
Dr. N. Shiri

_____ Examiner
Dr. D. Goswami

_____ Co-supervisor
Dr. V. S. Alagar

_____ Co-supervisor
Dr. M. Mohammad

Approved by _____
Chair of Department or Graduate Program Director

_____
Dr. Christopher W. Trueman, Interim Dean
Faculty of Engineering and Computer Science

Date _____

Abstract

Context-aware Service Registry: Modeling and Implementation

Alaa Alsaig

Modern societies have become very dependent on information and services. Technology is adapting to the increasing demands of people and businesses. Context-Aware Systems are becoming ubiquitous. These systems comprise mechanisms to acquire knowledge about the surrounding environment and adapt its behaviour and service provision accordingly. Service-oriented computing is the main stream software development methodology. In Service-oriented Applications (SOA), service providers publish the services created by them in service registries. These services are accessed by service requesters during discovery process. For large scale SOA, the registry structure and the type of quires that it can handle are central to efficient service discovery. Moreover, the role of context in determining services and affecting execution is central. This thesis investigates the structure of a context-aware service registry in which context-aware services are stored by service producers and retrieved by service requesters in different contexts. The thesis builds on an existing rich theoretical service model in which contract, functionality, and contexts are bundled together. The thesis investigates generic models and structures for context, context history, and context-aware registry. Also, it studies state of the arts database technologies to analyse its suitability for implementing a registry for rich services. Specifically, the thesis provides a thorough study of the structures, implementation, performance, limitations, and features of *Key-Value*, *Documented Oriented*, and *Column Oriented* databases while considering options for implementing a rich service registry. Database models of contexts and context-aware services are discussed and implemented. The relative

performance of the models are discussed after evaluating the test results run on large data sets. Based upon test results a justification for the selected model is given.

Acknowledgments

I would like to express my sincere thanks to my supervisors professor Vangalur Alagar and Dr.Mubarak Mohammad for their guidance, teaching, and training through the learning process of this master thesis. It has been an honour working with them as they are great sources of knowledge and inspiration and role models for humility and patience. I would not have been able to accomplish my thesis without their educational and personal support. Both have a great understanding which makes them believe in others, motivate them, and give them chances. I would never ask for better supervisors. I have learnt a lot from them throughout the work of my journey towards the masters.

I would like to thank my parents who have been always sources of encouragement and support and for having trust on me. Their giving and caring cannot be compensated and thanking them is never enough to express the love and sincerity I have towards them. This love includes all my sisters and brothers that are always there for me. A special thank for my companion in this journey, my brother Ammar. I would not have been able to achieve this without him. In all struggles I had, I found him there standing by my side listening and supporting. I extend my deep thanks to his wife Duaa for her patience and support throughout the whole journey. A great thank for my nieces Wedad and Sarah and my nephew Yassir for adding joy and smiles to my life.

A great thank from the heart for all friends in Canada. Their support and caring are undeniable. They were not only friends but also a family who were there to celebrate my happiness

and cheer me up during my struggles. The kindness and great personalities they have make them very memorable. Undeniably, I learnt a lot from their experiences and guidance. I am so blessed to have them in my life.

A special thank for other great people who were helping from overseas and who have never been late to listen, advice and support during these years. The far distance and different time zones were never reasons for ignoring help and support. I cannot express how grateful I am for them. Special gratitude for my brother Dr. Mohammad Alsaig and my professor Dr. Wadee Alhalabi.

Moreover, I would like to extend my thanks to Saudi Cultural Bureau for the financial support and help throughout these years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In Service-oriented Applications (SOA), service is the main object of the system. Services are published by service providers and stored in a central service registry. Service seekers can browse the registry to discover services. Hence, the modeling and implementation of a service registry should facilitate efficient service discovery and provision. This thesis studies the design of a context-aware service registry in which services that are associated with context information are stored. The thesis introduces a generic structure for context-aware services. Also, it provides three specialized models for context-aware services using three different NoSql databases. Moreover, the thesis introduces generic models for context and context history. Finally, the thesis provides implementation and performance analysis.

Modern societies have become very dependent on information and services. Technology is adapting to the increasing demands of people and businesses. Context-Aware Systems are becoming ubiquitous. These system comprise mechanisms to acquire knowledge about the surrounding environment and adapt its behaviour and service provision accordingly. These systems are expected to provide services to users based on context preferences or rules rather than just providing general classes of services for all types of clients at all times. Hence, many

researchers were motivated to improve SOA systems by enhancing service discovery and provision to benefit from context-awareness. This thesis is a contribution to this field and introduces a design and implementation of context-aware service registry.

In SOA systems, three components deal with services. These components are service registry, service provider, and service requester. The service registry is the component that stores and manages services. It is responsible for service discovery and provision. In the literature, the Universal Description, Discovery, and Integration (UDDI) is a standard that outlines the specifications for storing service functionality descriptions. Most of service registries that are used in the industry are based on UDDI which considers only the functional part of service descriptions. It uses Web Service Description Language (WSDL) to define model services. WSDL includes *businessService*, *bindingTemplate*, and *tModel* information to describe services registered in UDDI [LGZ+05]. *BusinessService* includes general information of services. *BinidingTemplate* includes any information related to the locations where services are stored and methods for accessing them. One or more *tmodel* information is associated with a network address as an element represented in each *bindingTemplate* structure. Each *tModel* structure is used to describe and distinctly define a Web service [BCE+02]. XML language is used in WSDL and the data is stored in a relational database [LL09]. UDDI depends on static information about service functionality. It does not support or benefit from context-awareness.

Integrating context-awareness in SOA enhances service discovery and provision. Formalizing context-aware services and registries is an essential prerequisite for providing context-dependant services for consumers. However, only recently, a formal approach for formalizing 'context-aware services' was introduced by Ibrahim [Ibr12]. The building blocks of SOA systems are modeled as *ConfiguredService*. In *ConfiguredService* model, *Service* functionality and its non-functional properties are bundled up with *Contract* that includes *Context*. The service part encompasses all information that is highly related to the service including functional,

non-functional and attributes data. The contract part in *ConfiguredService* includes any information that is highly related to context and legal rules. The contract part is loosely coupled to the service in order to allow a service to have different contracts. This study [Ibr12] provided only a formal service model, mentioned a high-level structure of service registry to store *ConfiguredServices*, but did not discuss methods for service discovery and service provision. To the best of our knowledge, there is no other service model that uses context-aware service registry. Motivated by the need to provide context-dependent services and the lack of registries to support context-aware services, this thesis extends the work done by Ibrahim [Ibr12] on service modeling by providing a generic model for service structure. Also, the thesis introduces a novel service registry structure and provides an implementation for it.

Most of the existing service registries have been designed using either XML or pure XML. Examples of such systems include UDDI and xUDDI [LL09]. There is no work that investigated database models for service registries. After some investigation into database models we found that NoSql databases provide powerful methodologies for storing, retrieving and updating data with high performance and scalability. Also, Google uses BigTable [CDG+08] for its database, and Amazon uses Dynamo [DHJ+07]. Both Google and Amazon use NoSql in some manner. NoSql databases provide powerful capabilities to model dynamic and rich information. Therefore, we decided to use NoSQL database as a tool for modeling context, context history, and service registry. NoSql databases are mainly categorized into *Key-Value Stores*, *Documented Oriented Stores*, and *Column Oriented Stores*. Each model has its own specific features and advantages. This led us to use these models to model context-aware service registry and compare their abilities. We provide both abstract and concrete models for context-aware service registry, service, and contract with due consideration to implementation requirements. Context, contract, and *ConfiguredService* are rich terms. We define them before providing their database models.

1.1 Contribution

The contributions of this thesis are:

- Generic structure for context, context history, and context-aware services.
- Generic structure for context-aware service registry.
- Three types of NoSql models and implementations for service registry.
- Comparative study and analysis of *Key-Value*, *Documented Oriented*, and *Column Oriented* service registries.

The work done to achieve these contributions is outlined in this thesis as follows. Chapter 2 provides a quick literature review in order to highlight the original contributions of this thesis. Chapter 3 discusses context definition and its types. Then, it introduces a new generic model for context and context history. In Chapter 4 we extend the formal service model of Ibrahim [Ibr12] and provide a new generic model for each of its components. A novel service registry model is introduced in Chapter 5. Chapter 6 introduces our three NoSql models for context-aware service registries and describe their implementations. Chapter 7 provides a comparative study and performance analysis for each implemented model. The thesis is concluded in chapter 8 with a summary of the work done and an outline of the future work.

Chapter 2

Literature Review

There exists a large body of literature on service-oriented computing and contexts. Since we are interested in database modeling of registries of context-aware services, we focus on the literature of service registry modeling in general, with or without context information. We highlight their approaches and outline their limitations.

In SOA, a service is an object that is stored in service registry, published by its service provider and discovered by service requesters. The service registry is the component that is responsible for storing services and providing the interfaces for service provision and discovery. This makes the service registry a complex component that needs to be modeled with good software architecture principles.

Many researches have proposed UML-based design of service registry [HL06]. Universal Description, Discovery and Integration (UDDI) is one such registry that has been in the market since year 2000 [web]. It stores service descriptions with references to their source locations in the repository, where actual services and application software are stored. The structure of a service includes only its function description. It does not include non-functional description, quality information or context. A user query can only mention the functionality, and service discovery is a simple matching of the query information with the specific service information

stored in the registry. To overcome this limitation, a modification to UDDI structure has been proposed [LGZ+05]. This modification assigns additional storage to UDDI in which non-functional attributes corresponding to each service in the service registry are stored. A user query is directed to the registry if it is about service functionality and it is directed to the adjoined repository if it is about non-functional descriptions. UDDI is not an efficient service registry architecture [LGZ+05], and it does not have all logical operations which are required for comparing and ranking services [Min08].

In [LL09] another important angle of UDDI drawback is brought out. UDDI uses relational database to store its data, but XML code is used for exchanging messages. The protocol used for messages is Simple Object Access Protocol (SOAP), which uses XML code. This is the reason for the overload work that comes from managing XML data with relational databases as it requires a lot of data shredding to database tables and vice versa, which leads to a massive work in converting text to XML and shredding data to relational table. Excessive conversion operations might result in valuable data loss. Therefore, it was motivated to structure xUDDI that is based on pureXML. This study provides reasoning for using pureXML claiming that pureXML is the best solution. By using pureXML, the structure of the data is stored in the registry and in the database with no need for conversion or shredding. Also, generating a table from XML source becomes easier and with less work. This paper also provides a way for performing classification and authentication of services on pureXML hierarchies. However, pureXML has its limitations when using large and rich objects. Searching, retrieving part of a document, and updating a document can be expensive. A new methodology for indexing might be required to evade the costs that result from parsing XML to every query to specify which part of the document meets the determined search standards [SCA06]. Other limitations of UDDI registry are discussed in [Min08].

In the aforementioned studies, there is no consideration to context information. We discuss context in details in Chapter 3. For the present let us view context as the characteristic that

defines the status in which services are eligible to be delivered. Storing context information in UDDI will involve additional overhead to the system, namely including some other operations such as frequent updates and comparisons. Context might include information that is not only related to a service but also to any component in the system. Also, service functionality might be related to different sets of non-functional properties in different contexts. Having two registries, a main registry to store services and an adjoined repository to store non-functional properties, causes a heavy load of data transmission operations between the registry and the adjoined repository. Therefore, modifying the UDDI structure by imposing context on the structure is not practical. Thus, the available approaches are inadequate to fulfil the requirement of a context-aware registry.

A new architecture for context-aware service registry is needed for improving service discovery and provision. In general, an eligibility criterion for service provision is related to a specific situation. In these situations the service functionality may not change but non-functional properties of the service and rules for providing the service might change. These situations are the contexts of interests for service delivery. A service requester must fit in one of the available contexts to be eligible to receive a service. Storing contexts improves service provision by adding intelligence to the system to specify which service specifications is eligible for that requester plus the ability to adapt the execution of service to some contextual information. Also, context-awareness enhances service usability by having the ability to use one service object in various contexts. These context-aware services need a registry that has the ability to store, publish and discover them based on context. However, very few studies in the literature attempted to provide a structure for a new registry to store context-aware services. We discuss them next.

In [CLC10] four types of contexts are introduced. They are

- S-context type is for single services contexts,
- C-context type is for community of services which is dynamic,

- U-context type for users, and
- W-context type for queries.

The paper proposed a new organization for the old UDDI Business Registry (UBR). It is aimed to structure services in a tree-structure based on similarity among them. A group of strongly similar services create a community. The focus of the paper is only on structuring the services that are strongly similar. Then the paper introduces querying with context. However, a complete structure of the registry is not made clear. The paper does not discuss the more important issues such as how contexts and services are related, how a service is mapped to different contexts, and services are discovered and published based on contexts. There is increasing demand to know how to structure registry with support for context-awareness which is not dealt with in this paper.

Another modification to UDDI has been proposed in [Lee08]. They propose to integrate Web service registries with web service quality management system (WSQMS). The goal is to reduce the effort of measuring and testing services on service providers and making web services more reliable. However, service reliability is not just related to service quality. There are international laws and trade rules that assure service providers and requesters rights certain . This means that the service quality will be constrained by these factors as well. So, service descriptions should have more information to store these laws and rules, which makes service object more richer. XML which is used in UDDI extensions are not suitable for describing complex objects. Besides, there is no clear structure for what information is to be included to describe service quality and context.

There exists a large body of literature in the study of context. Here we have chosen some recently published work on context modeling to compare our work. A full discussion about context, its formal representation, and modeling are taken up in Chapter 3.

In general, to our knowledge there is no report on database models for managing context

and context history. The UML context model [SB05] considers atomic and complex contexts. An atomic context is modeled as a class in which the two attributes are the name of the context and the source name of context. The only attribute of the complex context is the aggregation of its different contexts, with some logical operations. The two context models are independent of service. However, there is a class, called context-awareness, which is a component of the service. There is no semantics given for context-awareness. Their context model is both abstract and at best incomplete. It is abstract in the sense that the authors did not provide any language or database support that are necessary for implementing the model. It is incomplete in the sense that the type information necessary to capture the heterogeneity of information, the nature of context (permanent or temporal), and rules for using it in services are not modeled. Although the authors in [GS07] claim to have put forth a context-aware service application, the work does not provide any view of the context structure and how it is defined. Actually, the work is an extension to [SB05] in which they considered state-based and event-based contexts. On the one hand, a state-based context includes data of attributes that could be entity, device or user related. On the other hand, the event-based context encompasses a bunch of entity events. These events could be related to an application or a user with consideration to the history of events. However, there is no elaboration for how the context is structured and where the data is stored. Also, there is no specific structure for the history and what data could be included. In [TKS+10], the authors have introduced a context structure mainly for *Mashup* application requirements. They have considered the dimensions *when*, *where*, *what*, and *who* to construct contexts. The structure assigns several entities for each dimension. This makes context structure complex. In general, not all applications require the same context information. Consequently, their model could result in aggregating useless information. The model does not provide any mechanism to add new dimensions. Although they mentioned context change history in the paper, there was no information regarding history structure, model or attributes and data of the history.

2.1 Summary

We studied previous works for modeling context-aware service registries, service structures, and contexts. We found that there is no prior work that have provided a structure and implementation for context history and rich context-aware service registry. This motivates our focus on modeling and implementing a context-aware service registry. In the next chapter, we define context, discuss its different types, and review its history. Also, we provide a generic context structure that is ready for implementation.

Chapter 3

Context

As technology evolves, the dependency of society on the technology becomes more intense. This increases the need for smarter systems that can provide specific services rather than general ones. Service in the Health Care sector is an example. As a result many service-oriented systems have become pervasive, requiring context for service provision. *Context* can be either a location of a subject or any environmental surrounding such as temperature or weather affecting the subject. Much research is being done on context-aware computing, however very little effort has been directed to context modeling which is fundamental for developing context-aware applications. Context modeling should emphasize parameters and other needed elements for storing and managing a large number of contexts. Since context information, both past and current, are important for many applications, there is a need to manage them efficiently. Since this thesis is concerned with the design of context-aware service registry, we will first review how context is defined and comprehended in the current body of literature, and following it we will discuss context modeling. In particular we provide a generic structure for contexts that can be embedded in any Service-oriented Application (SOA) or any other ubiquitous computing system. Additionally, we propose a design for storing context history that maximizes data management and enhances accessibility.

3.1 Context Definition

In Oxford English dictionary context is defined as “the circumstances that form the setting of an event, statement, or idea, in terms of which it can be fully understood”. According to this definition, context is necessary to fully comprehend a statement, and hence it is different from the information conveyed through the statement. In Merriam-Webster dictionary context is defined as “the situation in which something happens, the group of conditions that exist where and when something happens”. This definition implies that context is the mutual relationship between the many conditions that exist in a situation in which an entity exists or an event happens. These two definitions are not exactly equivalent, because the first one is declarative and separates context from what is “uttered” (or happening), whereas the second definition makes context a logical entity. Both definitions are rather broad and do not suggest how to invent the elements to construct a context. The essential conclusion is that context is a *rich* concept and is understood in natural language communication. Therefore, it becomes necessary to refine its definition in order to be applicable for computing applications.

Context is implicitly understood by linguists [Rud56], philosophers [CC81], and AI researchers [ML90], [AS96], and freely used to express profound statements. There exists a large body of literature on context, and many definitions proposed by different researchers can be found in [BBH+10]. From computing perspective the first to propose a practical way to model context was Schilit et al [SAW94]. They proposed that “three important aspects of context are where you are, who you are with, and what resources are nearby”. Thus, in context-aware applications, *Context* is a meta-information that qualifies either data or information or an entity of interest in the system. For example, "*Alice is the caregiver for John*". By stating “when, where, what and why” such a care is given, we are adding meta information, that is a context for the service provided by *Alice*. Within SOA we can regard *Context* as any element that could affect the service provision and execution operations. Another simple example of

context mentioned in [Kei08] is that "the location of a subject is the context" which will decide whether or not a mobile service could be provided to that subject. Therefore, *Context* is defined as any environment element that gives rise to a meaningful interpretation of a function computation [Kei08].

Dey et al [DAS01] have given a definition of context that captures some aspects of dictionary definitions and the definition of Schilit. Their definition is "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application." This definition has been adhered to by researchers in Human-Computer Interaction (HCI). The key aspect of the definition is "relevance", which allows the developers to choose the parameters that suit the application, including mobility. However, these researchers have used only examples and informal notations to represent context.

Wan [Wan06] has given a formal representation of context. This representation can accommodate all definitions above, and more importantly it is supported by relational semantics. *Context* is defined as a collection of ordered pairs (d, v), where d is a dimension, and v is a value from the type domain associated with dimension d. Dimensions "Who, Where, When, What, and Why" have been identified as primary dimensions to construct any general context. A tag set, which is typed, is associated with each dimension. As an example, along the dimension "Where" the tag set can be "the set of city names or streets", and along the dimension "When" the tag set can be "the set of discreet time points". Therefore, context is a multidimensional typed entity. An example of a context in this representation is $[Who : Alice, Where : Montreal, When : 11 : 00]$. This context qualifies some events that might be experienced by *Alice* in $Montreal$ at time 11. More specifically, this in SOA can be the service delivery context for *Alice*.

The context representation of Wan [Wan06] is used by Ibrahim [Ibr12] for defining a *ConfiguredService*. We discuss the structure of *ConfiguredService* in Chapter 4. For the present,

it is sufficient to understand that *ConfiguredService* includes context information. This context is split into *ContextInfo* and *ContextRule*. *ContextInfo* is the context representation introduced by Wan [Wan06]. *ContextRule* is the service qualifier rule that has to be met to getting the service. Ibrahim [Ibr12] has explained the necessity of a formal representation of context for formally evaluating *ContextRule*. Such a formal evaluation can be automated, which is an advantage to justify service delivery at specific contexts. One of the aspects that was not discussed by Ibrahim [Ibr12] is *context types*. In modeling contexts for different applications it is essential to categorize contexts, based upon context information which may designate a context to be permanent, transient, or temporal. One of our contributions is the introduction of context types in context modeling in order to enrich the *ConfiguredService* model, which in turn will help implement more precise context-aware services.

3.2 Context Type

The three important entities in any SOA are *service*, *service requester* (SR), and *service provider* (SP). Each entity is influenced by its own set of contexts. Thus, we define the three context categories *Service Context* (SC), *Service Requester Context* (SRC), and *Service provider Context* (SPC). A context qualifies the status of the requester SRC while requesting or receiving the service. A SRC context includes information that is related only to SR. For example, the location and time parameters characterize the context of a client while requesting or receiving a service. A context of type SPC qualifies service availability and service quality for a service provided by an SP. This means that the information included in a context of this type is related to the SPs and their services. As an example, a SP may have license to provide services within 10 km of the location where SP is registered, and a SP context will include this information. A context of type SC is the most important one because it is to describe the service status. A SC context includes information that defines the eligibility and the availability for delivering the service.

Such information could be service related, provider related, or requester related. That is, SC is a combination of SRC, SPC and specific dimensions related to the service itself. Examples include contexts that restrict services available in a zone, to section of people in a zone, or time constraints for service availability. Context information in a SC context may sometimes overlap with contexts in SPC and SRC types. For example, "Movie Downloading Service" may be restricted to certain age groups which varies from one country to another. Hence the SC for this service can be represented as $[SRAge :\geq 18, SPlocation : USA, SRlocation : USA]$. This context restricts the service to a requester who resides in USA, and is at least 18 years old. The service provider location is USA. Thus, we need to include Age and Location attributes in SRC context and SP Location in SPC context.

Contexts from these categories regulate and restrict service provisioning in SOA. Contexts of SRC and SPC types must be pre-defined in the system, However, contexts of type SC may vary dynamically due to the mobility of SCs. In general, a SC type can be put into one of the three subtypes *permanent*, *temporal*, and *transient*. A permanent context needs to be saved. SC type contexts arise frequently in Health Care service domain. Diabetic patients need to know the sugar level at specific times of the days. They also need to know when exactly the sugar level is increasing, or peaks to a high or is decreasing to its lowest level. The contexts of blood test service might include information on time/day of reading, medication and its level, and other medical factors. Keeping the history of these contexts will help medical professionals as well as patients. A temporal context is one which may undergo changes. Many contexts that arise in business applications are of this type. As an example, a business rule of a multinational corporation might change depending upon the government imposed legalities. The instant at which a business rule changes is the instance in which a new context is created for enforcing the new business rule, thus overwriting the old context. A transient context is one that arises momentarily, and after its use the context may not arise again. Contexts that arise in many game playing applications are of this type.

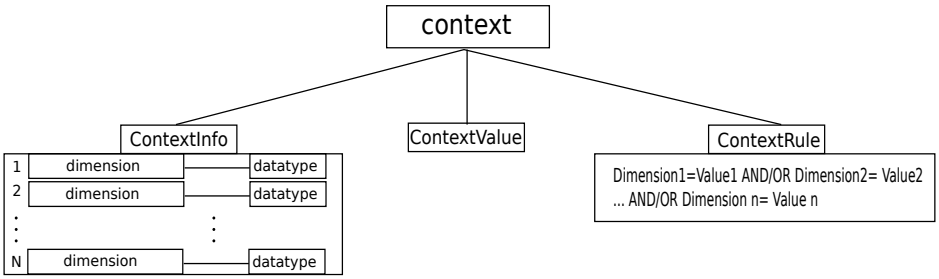


Figure 1: The Generic Context Model

3.3 Generic Context Model

Motivated by the structure of *ConfiguredService* we came up with a generic context model. In this section, we provide an implementation-oriented structure for this generic model. We discuss *ConfiguredService* in detail in Chapter 4. For our discussion here it is sufficient to assume that in a *ConfiguredService*, context has the two parts *ContextInfo* and *ContextRule*.

In our model we include *ContextInfo* and *ContextRule*. We also include an additional element called *ContextValue* in which information on the context collectors and values of current context are specified. The structure shown in Figure 1 illustrates this. In this *ContextInfo* field, we include dimensions and their type value and explicitly introduce the data type of the value. In the field *ContextRule*, a logical formula is included. The field *ContextValue* requires a more sophisticated structure in order to capture the change of values. We decided to include information such as the identifier of the context collector and the date and time of collection. Assuming that the information of the service context is gathered once the service is selected by SR. When the service is to be executed the system validates SR,SP eligibility for executing the service basing on the rules defined in the system.

The information included in the *ContextValue* is provided in two different nodes, as shown in Figure 2. Information in a dimension node is specific to each dimension. This information includes source ID which is the context collector's identifier. Since information for each

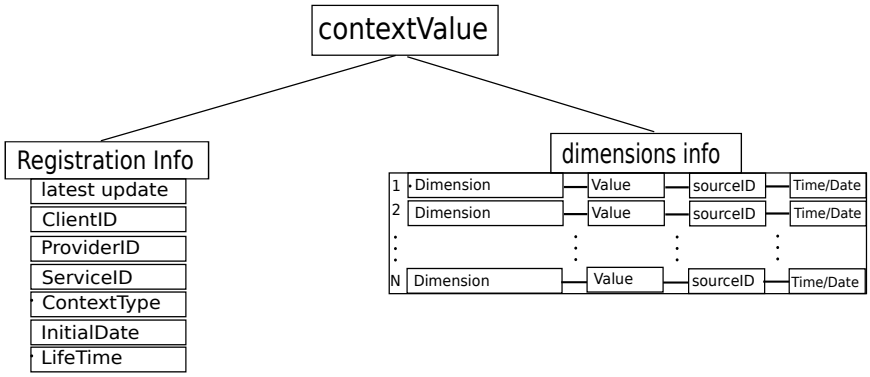


Figure 2: The structure of Context Value

dimension can be collected by several collectors, it is important to know which collector has collected the information in order to track it in case of a failure. Also, date and time of collection are made part of *ContextValue* in order to record the history of change. The second node of *ContextValue* is the registration node. This node includes information that is general for all dimensions such as context type, requester ID, provider ID, service ID and date/time of the last update. This information except date/time of last update, is not updated frequently. Rather, they are set when the service is executed and will remain the same for other updates. The fields in this node are defined below.

- lastupdate: includes the date and time of last update of the *ContextValue*
- requesterID: includes the ID of the requester to whom the service is provided
- providerID: includes the ID of the service provider
- serviceID: includes the ID of the service
- ContextType: can be permanent, temporal or transient
- initialdate: includes the date and time when the context was initialized
- lifetime: includes the time window for the life of the context

Example 1 *As an example of context, let us assume that a service provided by SP is limited to people who are maximally 150 Km away from the supplying station. The* ContextRule *states for this SP is*

- *if the SR is within 100 Km, then the service is provided with speed 15 mbps,*
- *if the SR is at a distance more than 100 Km but less than 150 Km, then the service is provided with speed 10 mbps, and*
- *if the SR is at a distance greater than 150 Km, then the service is not available.*

This rule can be formally stated as follows:

$$((d \leq 100) \rightarrow (s = 15)) \wedge ((d \leq 150 \wedge d > 100) \rightarrow (s = 10)),$$

where d is the distance between the SP and SR, and s is the service speed. In Table 1 location context uses d as 'dimension for distance', time is represented in t, source of information ID is shown sr, and shows the context-dependent service speed s for five different SRs.

Service Requester	Context Value	Service Speed
SR1	$[d:70, t:5:30, sr:sr_1]$	$s = 15$
SR2	$[d:100, t:9:30, sr:sr_2]$	$s = 15$
SR3	$[d:101, t:9:36, sr:sr_1]$	$s = 10$
SR3	$[d:150, t:5:30, sr:sr_1]$	$s = 10$
SR5	$[d:170, t:12:30, sr:sr_1]$	NA

Table 1: Context for five different SR and Service Description

■

Example 2 *In Example 1 the SR's are static. The same context rule can be applied to a service requester SR who wants the internet service on a smart phone device. Then, depending on the current location of the client the quality of service will vary. Because of mobility, the system has to track the location of the client in order to maintain or change the internet speed. Table 2 shows the context dependent service delivery on a mobile service of a single client, assuming that the client is*

in different locations at different times. We use d for 'distance dimension', T for 'time dimension', and SR for 'SourceID'.

Service Requester	Context Value	Service Speed
SR1	$[d : 70, T : t_1, SR : sr_1]$	$s = 15$
SR2	$[d : 100, T : t_2, SR : sr_2]$	$s = 15$
SR3	$[d : 101, T : t_3, SR : sr_3]$	$s = 10$
SR3	$[d : 150, T : t_4, SR : sr_4]$	$s = 10$
SR5	$[d : 170, T : t_5, SR : sr_5]$	NA

Table 2: The context tracking for a single SR

■

We attach Figure 2 in Figure 1 to obtain the full context model. This context model is sufficient to model all context types. In modeling a context of type SC we remark that information from the context of the service provider and information from the context of service requester are to be absorbed. The service provider context is known at service publication time, whereas the requester context will be known only at service discovery time. Consequently, the SC model will only be partial at the time of service publication. It is completed when a service requester discovers the service and ready to consume it. Example 3 explains these two stages.

Example 3 *Let us consider a service provider SP who provides an Internet service in New York city. The service provider context SPC has dimensions* $Location$, $Name$, *and* $Address$. *Let their types be respectively an enumerated type of city names, string, and record. Let* $C = [Location : NewYork, Name : ABC, Address : 15, fifthavenue]$ *be the SP context. Assume that the context rule for service provision is that a client must be at least 18 years old and live within 50 Km of the service provider location. So, this rule is written as* $(Age \geq 18) \wedge (Distance(x, y) <= 50)$, *where* x *is the address "15, fifth avenue', New york city", y is the address of SR, and* age *is tag value of dimension* Age *in the service requester context SRC. When the service is published by the*

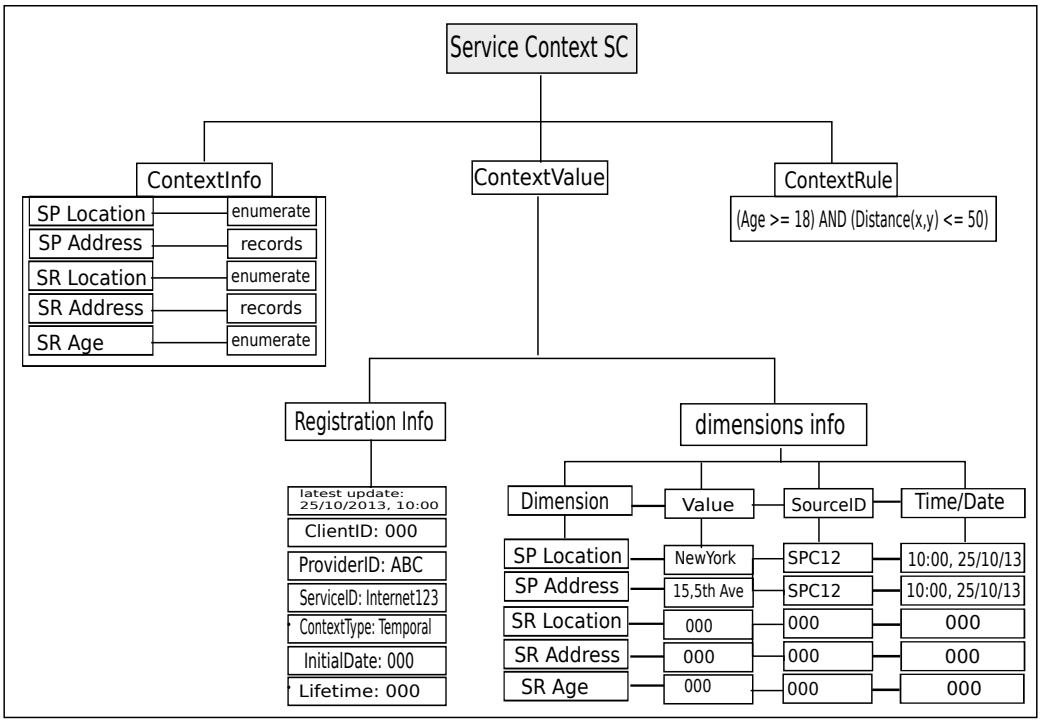


Figure 3: The Service Context Instance When No SR in the system

service provider, the service requester information is not known. Therefore, the service context is only partial, as shown in Figure 3.

When information of SR becomes available, the service context model becomes complete. As an example, let $C' = [ID : Alex123, Age :\geq 21, Location : NewYork, Address : 25, thirdavenue]$ *be the context of SR. This information is included in service context model when SR discovers the registry. At this instance, the service context model will be as in Figure 4*

■

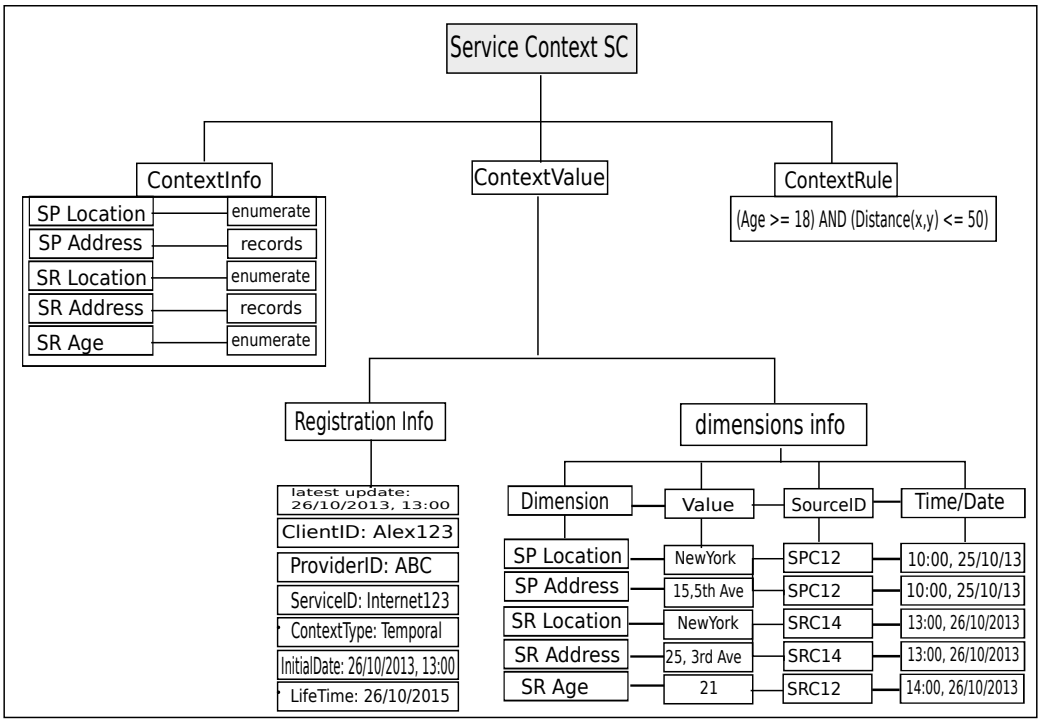


Figure 4: The Service Context Instance when SR is in the System

3.4 Service Context History Model

An analysis of historical information of contexts will provide valuable lessons to service providers in improving their business practises in future. Historical data regarding clients is very valuable for improving businesses and capturing the market needs and business trends. Through the accumulated contexts, service providers can observe and evaluate the services provided in the past and re-evaluate their business policies. In particular, service providers can perform data mining tasks for instance to discover the contexts in which the frequency of service requests peaked. When some of these contexts occur in future, providers can be better prepared to serve the clients. Also, historical information can be critical in health-related applications where there is an essential need to access the history of patients. For example, in providing health care for mental illness, it is very useful to investigate a patient's reactions in different medical situations

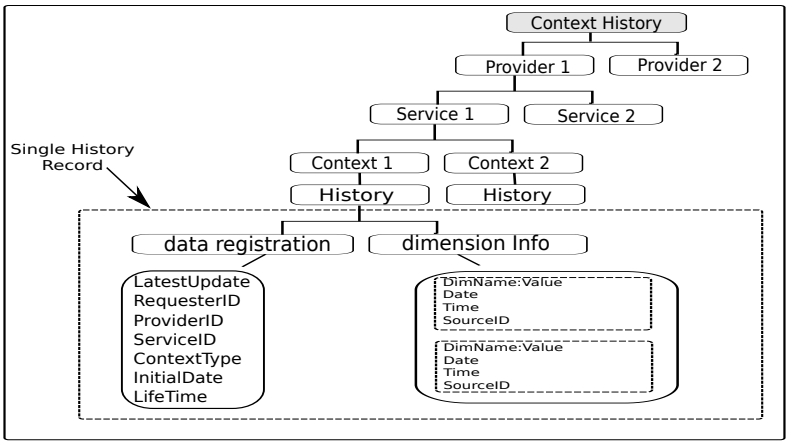


Figure 5: Context History Hierarchical Structure

for understanding and identifying the problem. The volume of data involved in historical evolution of contexts is rather immense. Consequently, we need a structure in which information can grow in an orderly manner, data access time is optimized, and insertion and deletion of information are done efficiently.

We propose a hierarchical structure that categorizes the historical contexts based on services associated with providers of the services. Figure 5 shows the hierarchy, where the subtree rooted at a service provider contains the services and the contexts of providing these services. Thus, with the help of information included in the data registration node, reaching the contexts of a specific service for a specific client can be made an easy process. Also, the hierarchical classification helps in keeping the growth manageable by narrowing it down to a specific provider, and service. Thus, the data related to one provider to one service is clustered together. Therefore, when providers are to access service contexts they only need to surf their own contexts among their own clients. This classification can be further refined by clustering the contexts for providers based on context types. Thus, permanent contexts are clustered together and remain untouched, whereas temporal contexts are visited periodically for updating the fields.

Additionally, to keep the history manageable we have introduced *lifetime*, *intialdate* and *contextType* fields in Figure 2. Based on *contextType*, a context is either to be deleted or retained.

In case the type of context is *permanent*, the context is stored. If the context type is *transient*, it is not saved at all. If the type is *temporal*, the lifetime field is added to the field *initial date* which will define the expiry date of the context. This expiry date is calculated whenever the clean-up process is activated and the record is deleted if either the current date information in it is equal or past the expiry date information. Medical application is an example of applications that require history storage.

Example 4 *Some diabetic patients need to provide monthly reports that includes 4-5 readings of their sugar level per day. This information is very important for doctors to specify the correct dosage of medication to be administered to the patients in order to stabilize their sugar levels. Therefore, it is essential to store the history of sugar tests done daily for each patient. There are medical devices that provide sugar level testing service. This device could be enhanced by automating it to produce reports. That is, the devices could be enhanced with contexts and to produce reports. The context dimensions for this service are Date, Time, Sugar-Level, and Status of Sugar-Level. The status could be N (Normal), H (High), or L (Low). The registration Information is not needed as we assume that the device is used by individual patients. Table 3 shows an example of a report of context history of a diabetic patient.*

Context		
Data | Time	Sugar-Level	Status
1/7/13 | 7:00	4	N
1/7/13 | 11:30	15	H
1/7/13 | 4:30	2.4	L
2/7/13 | 7:00	3.7	N
2/7/13 | 12:00	10	H
2/7/13 | 4:00	7	N
3/7/13 | 7:30	2.00	L
3/7/13 | 12:00	3.00	N
3/7/13 | 5:00	1.5	L

Table 3: Example of Context History of a diabetic patient.

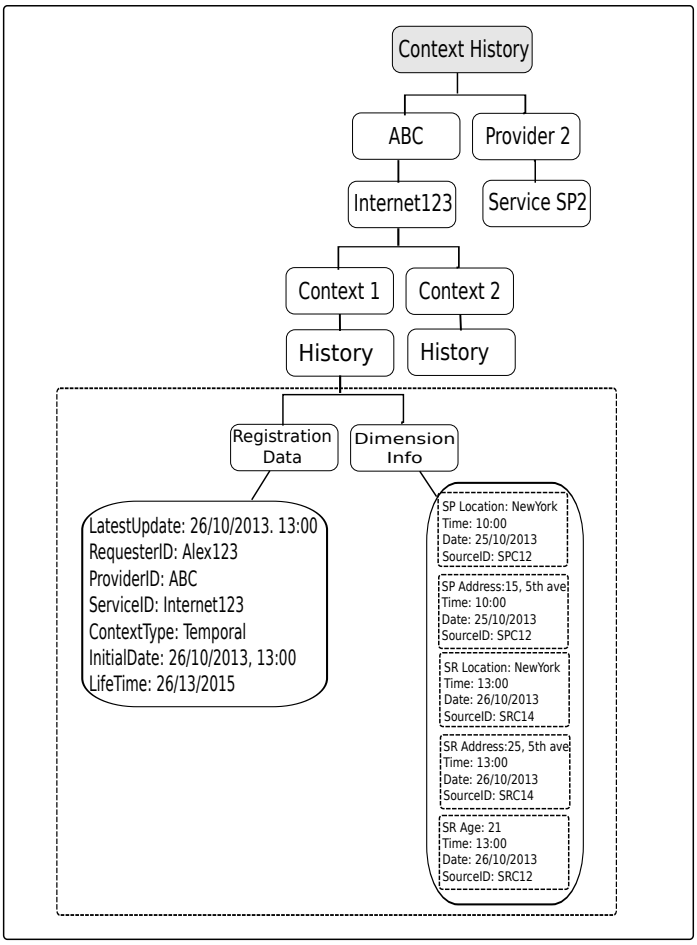


Figure 6: The Context History Instance of Internet Service

■

Table 3 presents a report produced by a single device at different contexts with a single function. When we consider a service provided to different SRs with different information at different contexts then we will have a hierarchy. Example 5 illustrates an instance of context history model for Internet service.

Example 5 *The Internet example in Section 3.3 showed context models at service publication and service discovery times. When a SR information is not available, the service context is only partially complete and hence context history is not there. So, in a context history model we do not store partially filled instance of service contexts. However, once the information of all service context*

dimensions is available we can start a context history. The latest updated information of context value stays with the context service instance until it is updated again. Once it is updated, the service context holds the new information and sends the older one to be stored in context history. Let us assume the completed context information of Example 3 as the initial SC context. That is,

$$SC = [SPLocation : NewYork, SPAddress : 15, 5thAve, SRLocation : NewYork,$$

$$[SRAddress : 25, 3rdAve, SRAge : 21]$$

Suppose it is updated with a new SR context information, $SC' = [SPLocation : NewYork, SPAddress : 15, 5thAve, SRLocation : NewYork, SRAddress : 110, 8thAve, SRAge : 30]$. *Then the initial context instance of SC (Figure 4) is represented in the context history model shown in Figure 6. If another SR uses the same service at the service context*

$$SC" = [SPLocation : NewYork, SPAddress : 15, 5thAve]$$

$$[SRLocation : NewYork, SRAddress : 10, 10thAve, SRAge : 31]$$

the information of SC' is sent to history. The updated context history is shown in Figure 7.

■

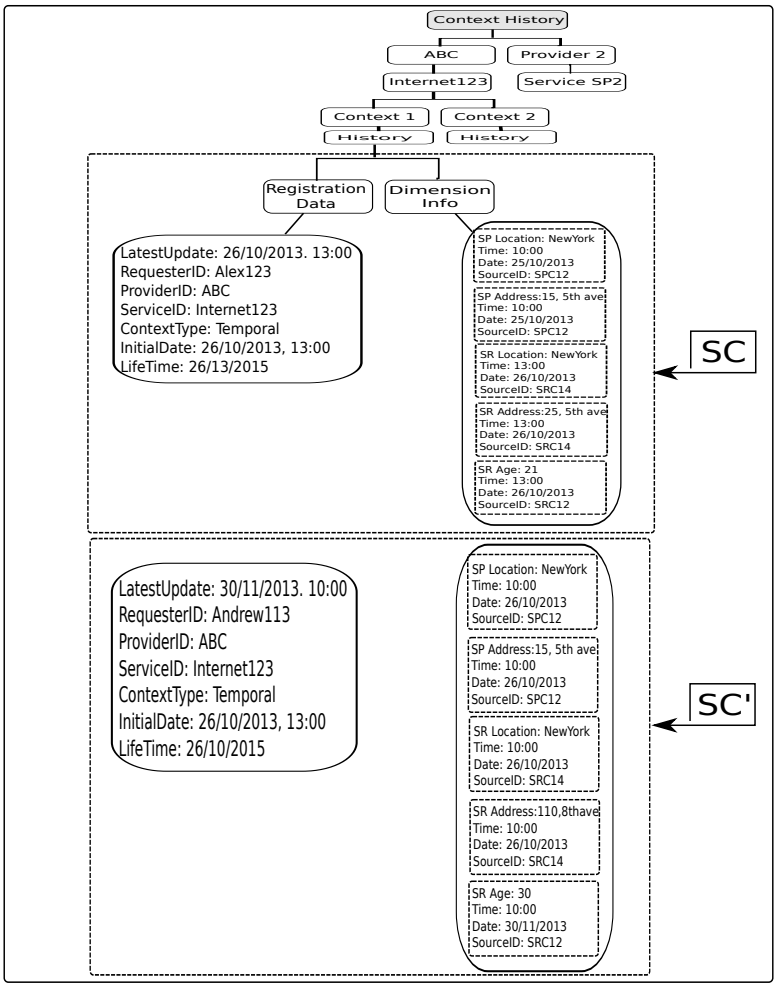


Figure 7: The Context History Instance 2 of Internet Service

3.5 Summary

In this chapter, context has been modeled in a generic way, and its dynamic characteristics have been explained. We introduced context types to manage dynamic context information in our model. A model for context history has been proposed. The proposed context types and history model are general enough that they could be used by different software systems that require context. In particular, SOA systems can embed the context model to provide context-dependent services. If a SOA system does not have context information as part of the service component, then embedding our models will have no adverse effect on their performance.

Chapter 4

Context-Aware Services

In many application domains services are to be provided by taking the context information of service requester into account. Health care domain is typical in requiring context-aware services. An important requirement is to sense the contexts of patients in order to provide proper services for them. In order that a service becomes aware of context, it is necessary to attach context information to service component. Thus, it is important that context specifications and context constraints are included in service definitions. Given this service structure, a service requester whose context is specified can be delivered the correct service. Motivated by this need to tightly couple service with context, Ibrahim [Ibr12] has introduced *ConfiguredService* in which a service is configured by context. The context information is included as part of *Contract* component, which includes *Trustworthiness*, *LegalRules* and *Non-Functional* properties. In this chapter, the *ConfiguredService* components are explained and clarified for our modeling and implementation needs. We extend *ConfiguredService* to provide a generic context-aware service model, which is ready for implementation.

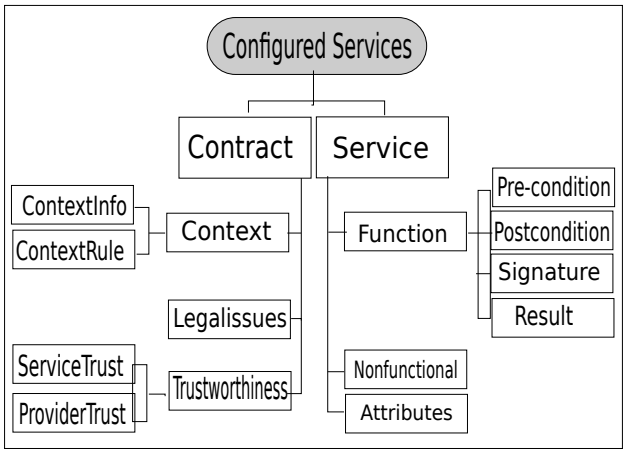


Figure 8: The Definition of the Configured Services [Ibr12]

4.1 Configured Services

The components of *ConfiguredService* are illustrated in Figure 8. *ConfiguredService* consists of a *Service* and *Contract* components. The former is the component that includes all data that are attached to service functionality and service description. The latter encompasses context as a parameter adding to it other parameters that depend and change when context is changed. These parameters are *LegalRules* and *Trustworthiness* properties. In the following sections we explain each component and provide a generic model for *Service* and *Contract* components.

4.2 Service

Service is the part that includes all the information that are essential to describe the service functionality and features. This service information is static and tightly coupled with *Service* component. That is, the service information can be changed only when new service (product) attributes are added. Any change of service information will not affect contract information. Likewise, the change of a context does not affect the service description. However, it does affect the parameter included in the contract. Therefore, *Service* separated from the context that is encompassed within the contract part. For example, a service description do not

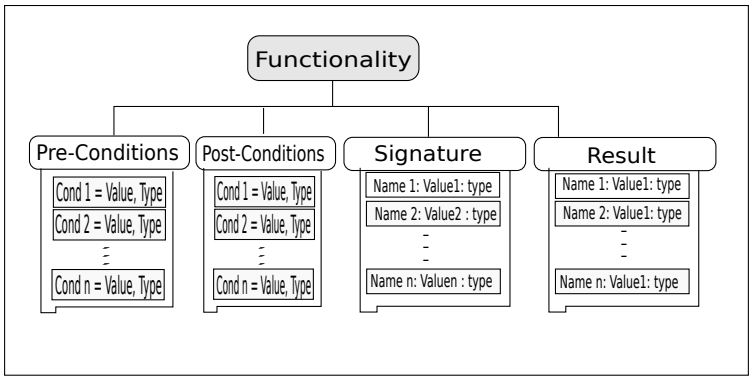


Figure 9: The Generic Structure of Service Functionality

change with the change of location. Hence, all service description is modeled as elements of service component. The service information is categorized into three parts: *Functional*, *Non-Functional* properties, and *Attributes*. To model a *Service* we need to model each part of it and put them together. We explain the steps next.

- **Functionality**

  The service function is defined by four elements. These elements are *Pre-condition*, *Post-condition*, *Signature*, and *Result*. *Pre-condition* includes condition that should be met by either service provider or requester to provide the service. *Post-condition* specifies what SP should provide and what SR is supposed to get. We need a model in order to allow their retrieval and modification, independent from other elements in the *ConfiguredService* model. Therefore, the *Pre-condition* and *Post-condition* elements are separately modeled as a list of values and their types, considering each value as a condition. The type helps with validation and executing processes. *Signature* includes information, such as address, that is unique to an SP. This information is different from a service provider to another. Hence, the model needs to have different identifiers and values for signature fields. Also, each identifier needs to have a defined data type. This is because *Signature* information is needed when the provider is requested to enter its signature for service execution or any other reason required during service provisioning process. The

type of data, in any element of service or contract, can be primitive type or complex. The complex type is defined by service provider during service publication and is stored in the system as part of the domain application. All the examples that we have used involve only simple types, however, any complex type defined by SP can be included in the model. From modeling perspective, each signature needs the ability to store a list of identifier and their values with their type. Each identifier represents the name of the files such as city, street name or location, the value includes the data for those fields, and the type is a data type of the entered value such as string, integer or a defined type in the system. Thus, for each signature element, the ability to store different keys, values and types should be provided. *Result* stores the information that is returned after executing the service. Although service execution is not within the scope of this thesis, our structure should provide *Result* element in which execution results are provided service requester. We model *Result* by a list of keys (identifiers) and values with defined types. The identifiers represent names of the returned value. The value field is not assigned till the service is executed. Types define the type of values returned from executing the service. The *Functional* component with all its elements are illustrated in Figure 9.

Example 6 *Let* Rent-Car be the function name of a car rental service. A precondition of this function can be

$$\text{Pre-Cond:} validC(creditCard) \wedge validD(DrivingLicense),$$

where $validC$ and $validD$ are functions that will validate the status of $creditCard$ and $DrivingLicense$. A postcondition for the function *Rent-Car can be*

$$(confirmCredentials = T),$$

Let the $Signature$ *information be* $(Address = (XXX, string)$*, and the result of booking Result be* $(bookingConfirmed = T)$*. This service information is represented in Figure 10.*

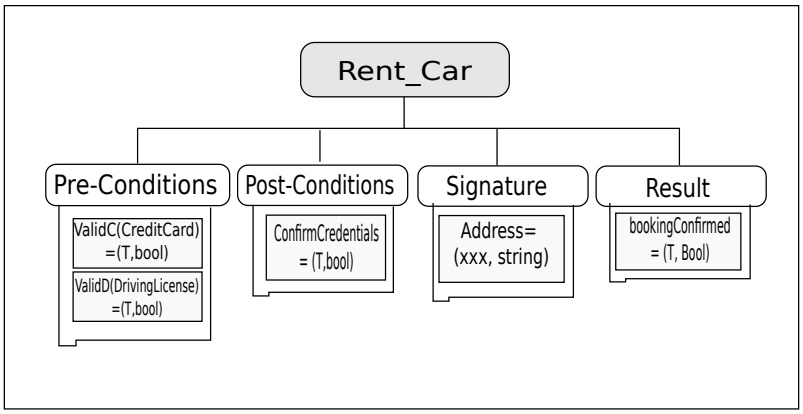


Figure 10: Example of Service Functionality Element

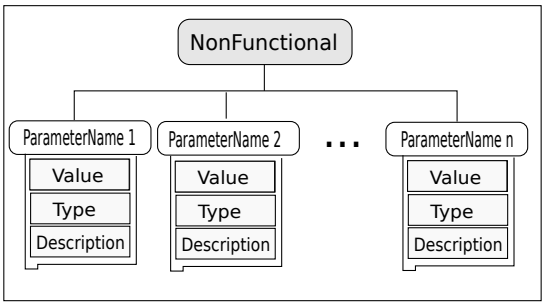


Figure 11: The Generic Structure of Non-Functionality Properties

■

- **Non-Functional Properties**

  Non-functional properties refer to the characteristics that are not related to the functionality of the service but are essential for its acceptance. For example, for selling a book, the price is a *Non-Functional* property. These non-functional properties are different from the attributes. This is because the attributes are to describe the service functionality, whereas, the non-functional properties are descriptions that are not related to the function but they are fixed and not related to the context. Therefore, these information included as part of the service. However, from modeling perspective the *Non-Functional* properties are treated similar to attributes. Each identifier is the name of the non-functional property

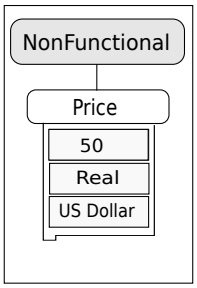


Figure 12: Example of Service Non-functional Property

that needs a value and data type of this value. Each non-functional property may need extra information such as currency type for price. Therefore, it is decided to add a description parameter for the identifier which will provide semantic information on the recorded data.

To model the non-functional part, we introduce four parameters. These parameters are *Name* which is a string, *Value* which is the value of the name parameter and its type is defined in the type field, *Type* which is the data type of the value, and *Description* which is a string value that describes the *Name* parameter. This information should be easily accessed and queried by service seekers. Figure 11 illustrates the *Non-Functional* model.

Example 7 *Consider the non-functional property* $price$ *for* Rent-Car service

$$Price : (value : 50perDay), (type : Real), description[(CurrencyType = USDollar)].$$

This information is modeled in Figure 12.

■

- **Attributes**

Attributes are specific characteristics of a service. Specific properties add another dimension to describe certain service aspects. Often the terms “service” and “product” are interchangeably used in certain application domains. For example, “Life Insurance (LI)”

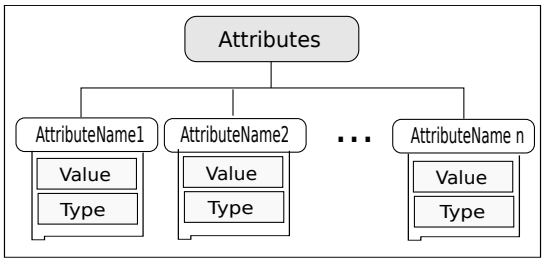


Figure 13: The Generic Structure of Service Attributes

is considered as a product in Insurance Industry, although we get only a service. The attributes of that service will describe specific features of a LI. In some other domains, product and service come together. In getting "mobile service", we buy a cell phone, which is a product, as well as telephone service using that cell phone. The same remarks apply to car rental services. However, if the service does not involve a tangible product, the service attributes are to describe the service functionality in some depth. As an example, an on-line bill paying service does not require the user to own a computer; however, the user must have access to the Internet. Thus, attributes are required to describe the service functionality and to characterize the products associated with a service. In modeling we found that the model is not affected by the service type. Therefore, we let *Attributes* model encapsulates *Name*, *Value*, and *Type*. The *Name* and the *Value* fields describe service and/or characteristics. However, the *Type* field is needed mainly to support transforming data between units of the system. That is, type is needed by other units to recognize the data. This is discussed later in the next chapter. Like non-functional properties, attributes information are accessed and queried a lot by service seekers. The model for attributes is illustrated in Figure 13. It supports data accessibility and retrieval.

Example 8 *For car rental example with functionality* Rent-Car both service and product (car) attributes should be modeled. Some of the attributes that are of interest to the

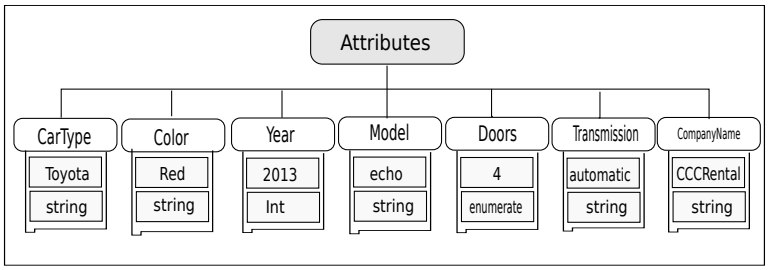


Figure 14: Example of Service Attributes Part

consumer are the following:

$$CarBrand : [Toyota, string],$$

$$Color : [Red, string],$$

$$Year : [2013, Integer],$$

$$Model : [Echo, string],$$

$$Doors : [4, Integer],$$

$$Transmission : [automatic, string],$$

$$CompanyName : [CCCRental, string].$$

The name of the attribute is the identifier and this key is mapped to its value and its type. Figure 14 shows the *Attributes* model for this example.

■

4.3 Contract

The *Contract* part in a *ConfiguredService* encompasses all the information that are changeable with some context and is loosely coupled with service. That is, service part can be the same while the *Contract* part is different for different *ConfiguredServices*. In other words, the same

service could be mapped to several contracts on which the service with each *Contract* creates a different *ConfiguredService*. *Contract* may change based on *Context*. That is, different locations may imply different legal rules. However, that does not affect the service whatsoever. Thus, the service and contract are separated while *Context* and *Contract* are joined. For example, a video download service could be provided in different countries such as Canada and England. Because, every country has its own restrictions and laws on some services, every country needs to have a different contract. For this particular service, the age restriction for downloading movies varies depending on the country. Therefore, every country could have a *Contract* that includes the legal rules and laws for this country. So, a contract for England is mapped to the video service and another contract for Canada is mapped to same video service which means that every country has it own *ConfiguredService*.

The contract part includes the three parameters *Trustworthiness*, *LegalRules*, and *Context*. Context part includes information to define service eligibility for a specific service requester. It includes *ContextInfo*, *ContextValue*, and *ContextRule*. Also, managing context history is important for many applications. Thus, context is considered the most complex part among the three parameters. Therefore, we have separately explained all information related to context definition, types, general model and context history thoroughly in Chapter 3. Below we give an example for context modeling based on the discussion in Chapter 3. Following it, we explain the models for the other two parameters.

- **Context**

Example 9 *The service context of car rental service includes information of the service provider SP and service requester SR. The SP provides a rule stating that the service should be provided to a person who is at least 18 years old. This rule is represented as follows:*

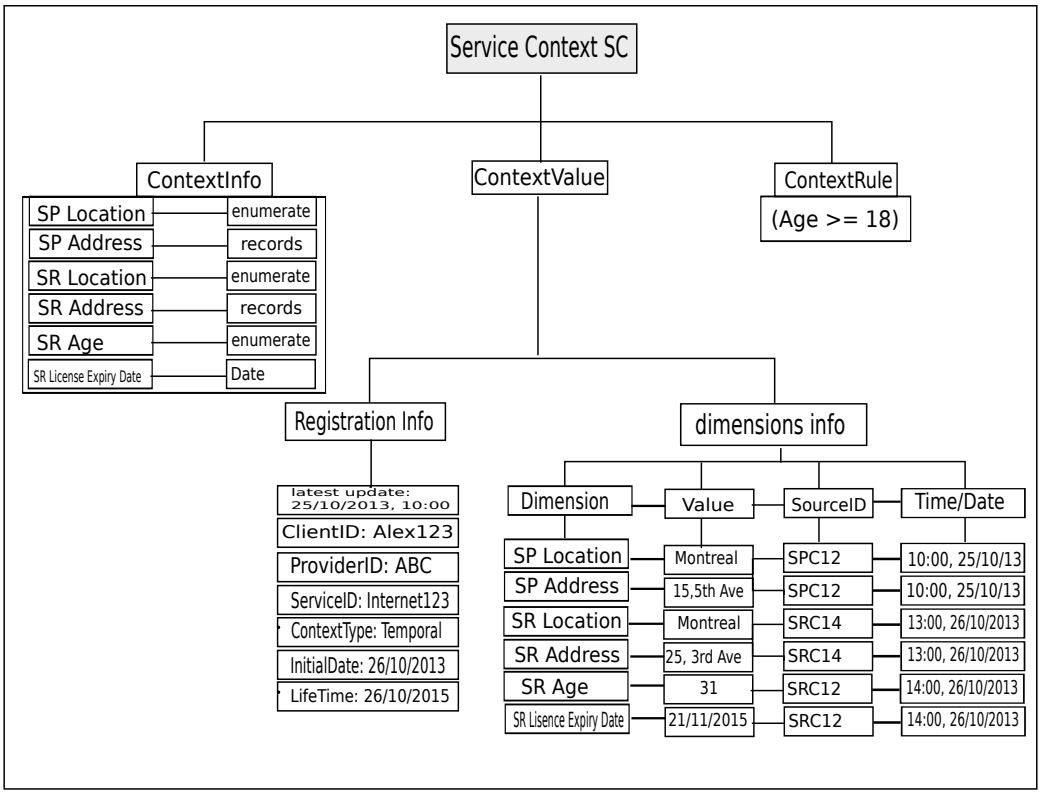


Figure 15: Example of Context Element

$ContextRule = (Age \geq 18)$ *The service provider context includes Location, Name and address information that is written as* $SPC = [Location : NewYork, Name : ABC, Address : 15, fifthavenue]$. *The service requester needs to provide Age, Location, Address and Driving License Expiry Date. The required information for SR is written as* $SRC = [ID : Alex123, Age :\geq 31, Location : NewYork, Address : 25, thirdavenue, DrivingLicenseExpires : 21/11/2015]$. *The context model is shown in Figure 15.*

■

- **Trustworthiness**

  Trustworthiness parameter stores trust information related to service and/or service provider. Thus, *Trustworthiness* is composed of *ServiceTrust* and *ProviderTrust*. *ServiceTrust* includes information that is related to service quality, such as timeliness and

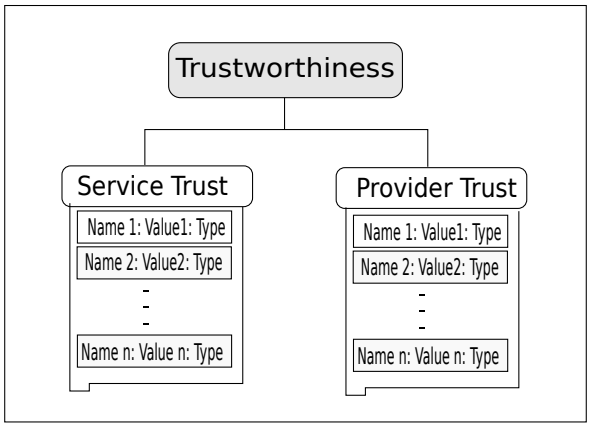


Figure 16: The Generic Structure of Trustwothiness

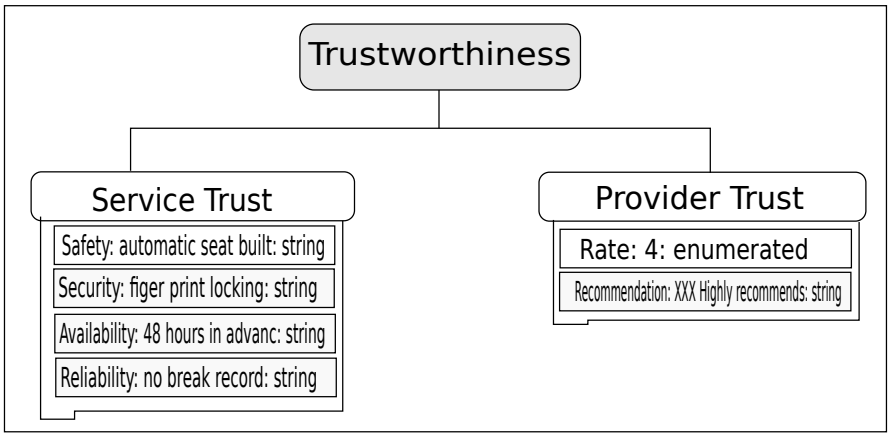


Figure 17: Example of Trustwothiness Element

safety. It is considered *ServiceTrust* to be composed of safety, security, reliability, and availability. Service Provider lists the claims of the Service Provider in some quantitative or qualitative terms. Service Provider includes trust recommendations of peers and reviews of clients. However, because of the generic nature of trust information, we decided to model both *ServiceTrust* and *ProviderTrust* as a list of names, values of those names and their data type . These names work as identifiers to their values. Figure 16 is the generic model for *Trustworthiness*.

Example 10 *Let the trustworthiness of a car rental service be as follows:*

$ServiceTrust : [(name : Safety), (value : automaticseatbuilt), (type : string),$
$(name : Security), (value : fingerprintlocking), (type : string),$
$(name : Availability), (value : 48hoursinadvance), (type : string),$
$(name : Reliability), (value : nobreakrecord), (type : string)]$

$ProviderTrust : [(Name : Rate), (value : 4), (type : enumerate),$
$(name : Recommendation), (value : XXXhighlyrecommends), (type : string)]$

The trustworthiness model for this example is shown in Figure 17.

■

- **Legal Issues**

  LegalRules are related to the business model and trade laws in the locations where services are made available. A few examples are refund rules, penalties for contract violations, and service requesters rights. To model *LegalRules* part, it is both necessary and sufficient to have the ability to store two string values for each rule that could be retrieved and compared to other string values. The first value called *Informal Rule* which is a textual rule representation that is easily readable by service requesters. The second value is *Formal Rule* that represents the rule formally to be understood and used by the system. It is assumed that the service provider enters both values during service publication. It is validated and analysed by the system before publishing the service. Considering modeling those values, both need to have a identifier. Therefore, each rule stores a name for the rule and two string values which are the the *Informal* and the *Formal* rules. The string type is proper and sufficient to represent both textual and formulated values. The model is described in Figure 18.

Example 11 *A few legal issues involved for car rental include information about collision*

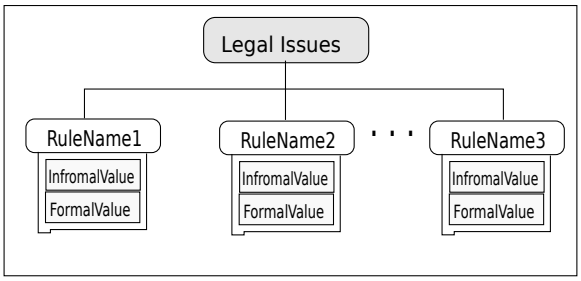


Figure 18: The Generic Structure of Legal Issues

insurance, parking violation, contract renewal, discount, and deposit. Each rule is represented in the form $IF\langle condition\rangle THEN\langle action\rangle$.

CollisionAndLiability **:**

($Informal Value$: *The collision is not covered, if an accident happened then the client CrditCard is charged*)

($Formal Value : IF\langle accident\rangle$

$THEN(CollisionCoverage = Null \wedge Liability = Null \wedge charge(CrditCard))$

ParkingViolation **:**

($Informal Value$: *Must be paid before returning the car, otherwise 100 dollars is additionally charged*)

($Formal Value : IF(ParkingViolation) \wedge \neg(DatePaid(ViolationFee) < (DateofReturn(car))$

$THEN(PayonReturn(RentalFee + 100))$,

where the functions (PayonReturn), (DatePaid), and (DateofReturn) should be evaluable in order to enforce this rule. So, in the case that the parking violation fee is not paid, the function (DatePaid) will return the value ∞ *which will make the predicate return a value.*

ContractRenewal **:**

($Informal\ Value$: *Automatically renewed, inform us in at least two days advance of contract end date*),

($Formal\ Value : IF(InformDate(user) + 2 < DateofReturn(Car)THEN(ConfirmRenewal))$

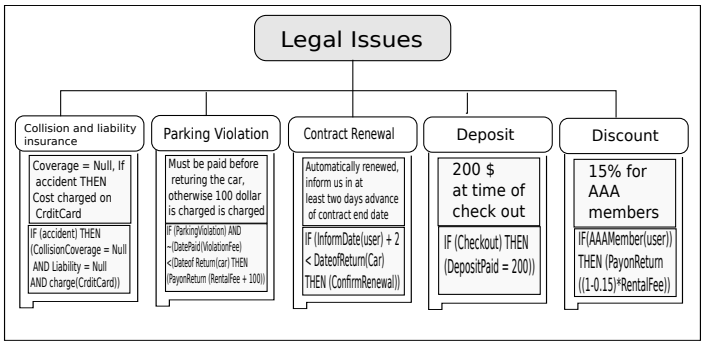


Figure 19: Example of Legal Issues Element

where (InformData) and (DataofReturn) are functions and (ConfirmRenewal) is a predicate.

Deposit **:**

$(InformalValue$: *200 dollars at time of check out*),

$(FormalValue : IF(Checkout)THEN(DepositPaid = 200))$

Discount **:**

$(InformalValue$: *15 % for AAA members*$)$

$(FormalValue : IF(AAAMember(user))THEN(PayonReturn((1-0.15)*RentalFee))$

The above legal issues or CarRent servic is shown in Figure 19.

■

4.4 Configured Service Generic Model

We put all the above generic models together to arrive at *ConfiguredService* generic model shown in Figure 20. In this model, the loose coupling between *Service* and *Contract* is explicitly modeled as a relation. The relation describes many to many relationship, using the traditional database definition. Because of the loose coupling between *Service* and *Contract* models, and the internal structuring within each model, it is possible (1) to apply changes on any parameter of any components, and (2) to navigate and retrieve data easily.

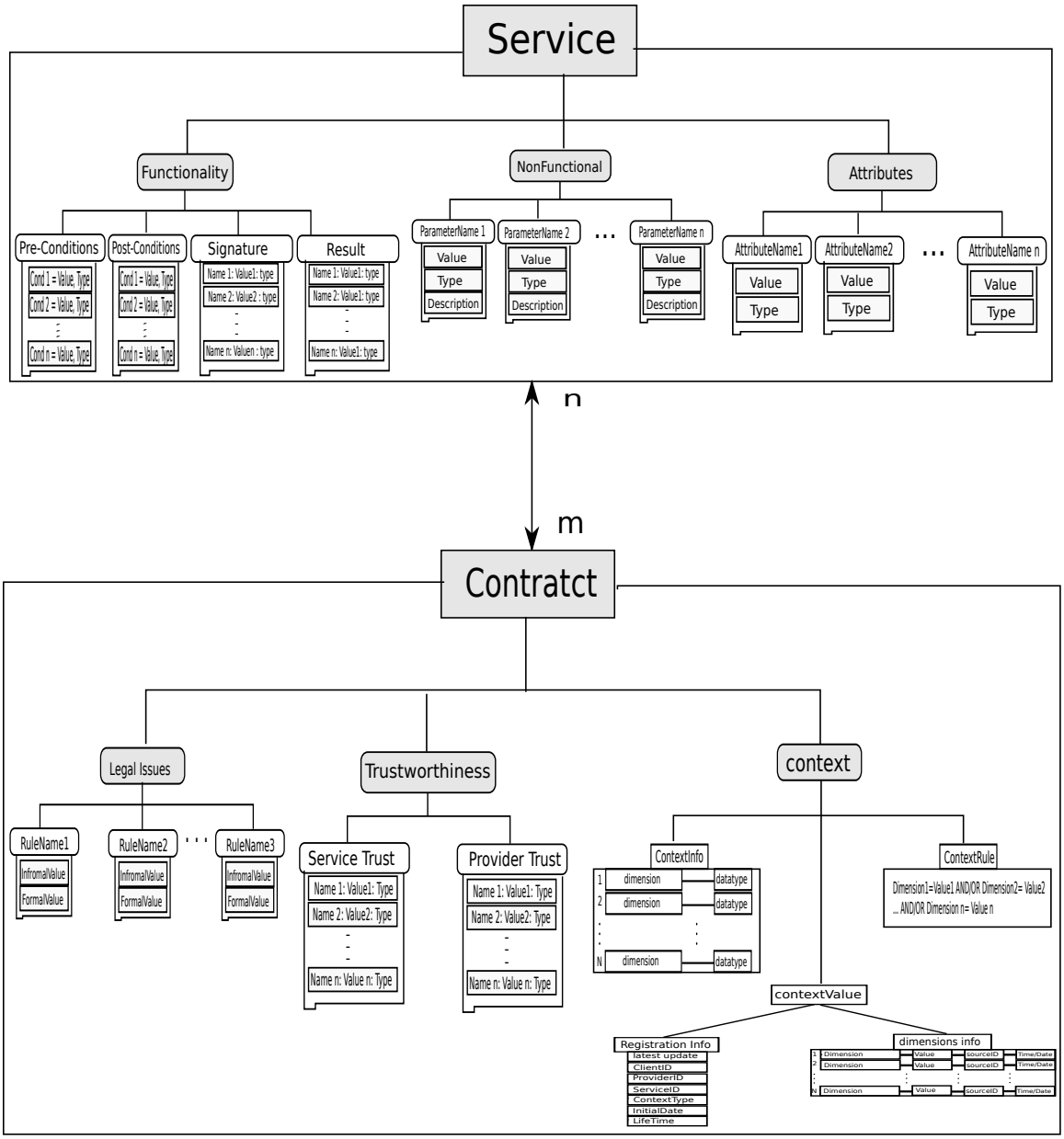


Figure 20: The Generic Structure of Configured Service

Example 12 *We take all the models for car rental developed in previous examples and put them together as required by the generic model. This results in the model for a specific car rental service shown in Figure 21.*

The relation between the contract and the service component is changed to (one to one) because particularily the carRent service component with this contract creates a single ConfiguredService.

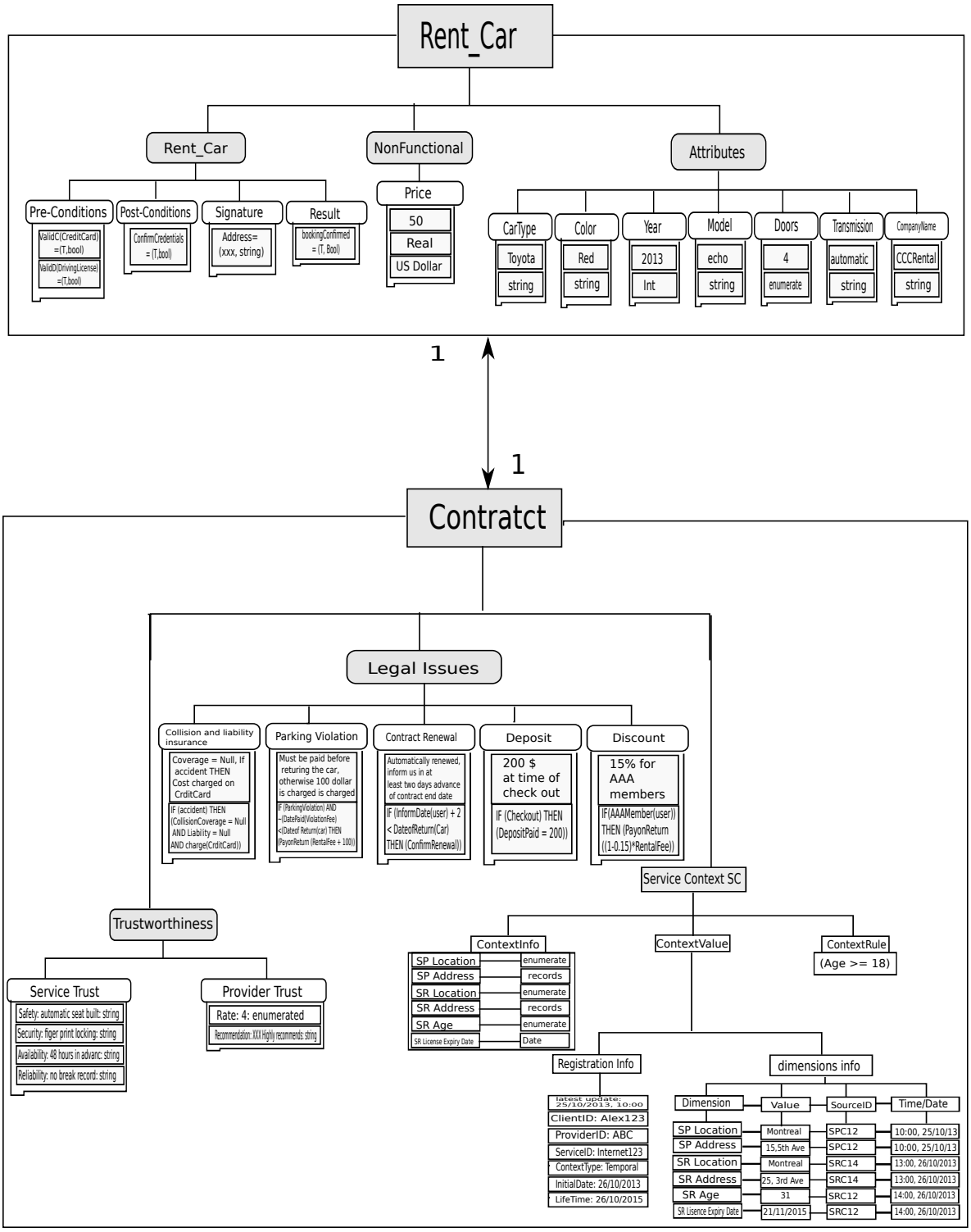


Figure 21: CarRent Configured Service Example

■

4.5 Summary

In this chapter, each formal element of *ConfiguredService* is clarified by explaining the data included in each element and the structure needed for each element. A generic model is provided for each element of *ConfiguredService* plus an example for each one of them. Finally, a general structure for *ConfiguredService* is proposed. As the *ConfiguredService* is constructed from all these components, assembling all the provided examples for each element constructs a complete example of *ConfiguredService*.

ConfiguredServices are stored and managed in a *Registry*. The *Registry* needs to be structured and modeled in a way that enhances accessibility and manageability. The next chapter discusses *Registry* that stores context-aware services and proposes a structure for categorizing, storing and managing *ConfiguredService*.

Chapter 5

Context-Aware Service Registry

Service registry is defined as *a specific type of repository that allows companies to catalog and reference the resources required to support the deployment and execution of services*[K+05]. That is, registries are used to store service descriptions and references to their resource locations. The information of the registry and the repository could be stored in one storage or different separate storages[Min08]. The current available registries, such as UDDI [BCE+02], store web services using WSDL [CCMW01]. However, these registries do not support storing rich services that include non-functional information and context. In Chapter 4, *ConfiguredService* is discussed and we outlined a generic service structure. *ConfiguredServices* need a registry that provides methods to support high manageability and accessibility for such rich services. The following features [Ibr12] are necessary for a service registry.

1. It stores *ConfiguredService*.
2. It enables service providers to publish and manage services.
3. It controls access using *Role Based Access Control (RBAC)*.
4. It provides domain knowledge and semantic methodology to be used by service providers and service requesters.

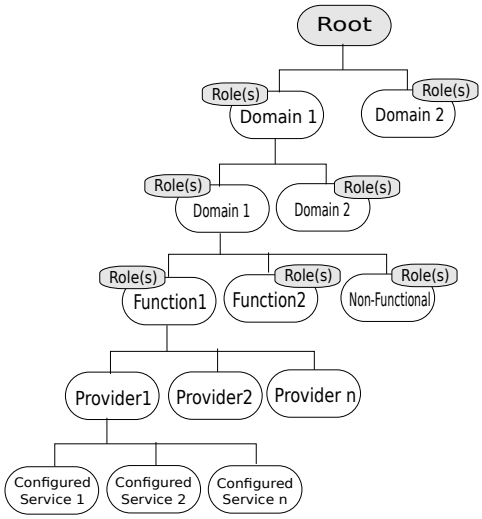


Figure 22: The main structure Service Registry

In this chapter, we introduce a novel generic structure of a context-aware service registry.

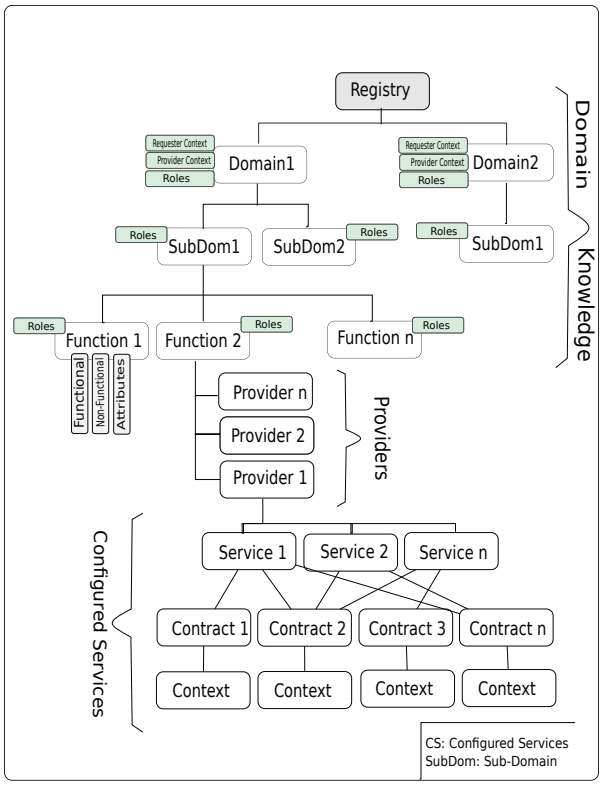


Figure 23: Context-Aware Service Registry Storage structure

5.1 Context-Aware Service Registry

An abstract structure for the registry storage is shown in Figure 22. Since *ConfiguredService* description includes context information and context rules for service availability, services stored in this registry can be discovered to best suit the contexts of service requesters. We expect that, in a service-oriented application in which this registry will be embedded, context-awareness will be part of service queries. We call the registry *Context-Aware Service Registry* (CASR).

CASR is structured as shown in Figure 23. It has three main parts. Below we discuss them and explain their models.

5.1.1 Domain, Sub-Domain and Function

The first three levels of the hierarchy are *Domain*, *Sub-Domain* and *Function*. They are used to classify the domain knowledge governing the information in *ConfiguredServices*. *Domain* represents a wide area of knowledge, such as Health Care, Transportation, and Entertainment. *Sub-Domain* is to represent a subset of knowledge within the domain. As an example electronic games is a sub-domain of games which in turn is a sub-domain of Entertainment. *Function* describes a type of service that is provided under this area sub-domain. An example of function is OnlineGameStores within electronic games sub-domain. A child of a function node stores information on providers of services with that functionality, and for each service provider what the *ConfiguredServices* are. In this section we discuss the structure in details up to the function level in Figure 23.

The nodes of these three levels share some elements, which we call *Followers*, *Followings*, *Level*, *NodeType*, *ChildType*, and *Roles*. *Followers* and *Followings* are used to build the relationship among the nodes of the CASR registry. *Level* is to determine the level in the tree hierarchy. This gives the system the ability to recognize nodes position of the tree as either root, intermediate node, or a leaf. *NodeType* defines the type of the node that includes this field. *ChildType* is to strict the children of a domain, sub-domain or a function, to either nodes from function type or sub-domain type. *Roles* are used to manage accessibility to *ConfiguredServices* by restricting domains, sub-domains and functions to specific user roles.

Before explaining each added field, we want to introduce the rules that restrict our DKN:

- the domains children are always sub-domain.
- the sub-domain can be further narrowed by a sub-domain once. That is, the DKN number of levels cannot be more than four levels. It is either three where the levels are domain, sub-domain and function, or four levels where the levels that are domain, sub-domain, sub-sub domain and function. This decision is based on the three clicks rule [RWC12]

which surveys cases and features that attract users to use one website more than another. One of the features that suits service requesters is that the number of clicks to browse a website preferably does not exceed three clicks. Hence, we decided to restrict the number of levels on a hierarchy of DKN to maximum 4.

- function cannot be a child of a domain, it is always a child of a sub-domain.

Level is a field of domain, sub-domain and function nodes that determines their level of the hierarchy. The levels is fixed with the following conventions:

- Level =0, this is for domain nodes only as they are the roots of the hierarchy. Once the system finds the value zero in this field, it recognizes that this node is a root node. The value of *Level* for domains nodes is restricted by this formula

$$IF((type = \ 'dom')THEN(Level = \ 0))$$

- Level = 1 or 2, this is for the sub-domain nodes as they are children of domain. Any node has the value 1 in their level field, it is a sub-domain node. However, a sub-domain can have another sub-domain as a child that narrows the category into a sub category. The sub-domain can present only in two levels of the hierarchy which are second (domain child) or third (sub-domain child). This will be explained in coming paragraphs. However, at this point, we provide the formula that restricts the sub-domain level number is assigned by the system following the methods below:

$$IF(type =' sub')AND(ParentType(ID) =' dom')THEN(level = 1)$$

$$IF(type =' sub')AND(ParentType(ID) =' sub')THEN(level = 2)$$

- Level = 2, 3, these values are assigned to level's attribute of function nodes. A function can be a child of a sub-domain node which results into assigning level field to value 2

($level = 2$). Also, a function can be a child of a sub-sub-domain node which results in $level = 3$.

The *Level* information of nodes are assigned implicitly to the nodes during construction.

$$IF(type =' fun')THEN(level = (ParentLevel(ID) + 1))$$

ChildType is to determine type of children a node can have. If $childType = 0$, then the children are from type sub-domain only. If the $childType = 1$, then the children of this node are all functions and other types are not allowed to be published under this node. The domain is always restricted to $childType = 0$ as domains children are always sub-domains. However, the sub-domain *ChildType* can be either $childType = 0$ or $childType = 1$. This is because the sub-domain node can have children from one type. Therefore, childType is determined for sub-domain node when the first child node is born. Depending on the first born child type, the other sibling of this node should have the same type. That is, if a sub-domain1 node is selected to be a parent of another sub-domain2 node, then other nodes of sub-domain1 should be sub-domains as well. In contrast, if the node gets a child from type function, then all other children of this node must be function as well. Function does need *ChildType* field as it is a leaf node of the DKN. The formal representation of assigning a value to *ChildType* field for sub-domain node is:

$$IF((Type =' dom')THEN(ChildType = 0))$$

$$IF((Type =' sub')AND(FirstChildType =' sub'))THEN(ChildType = 0)$$

$$IF((Type =' sub')AND(FirstChildType =' fun'))THEN(ChildType = 1)$$

NodeType is to tell the system the type of the node. This field is included in all nodes of the system that includes DKN nodes plus Service Provider, *Function*, *Service*, *Contract*, and *Context*. The type value could be dom for domain nodes, sub for sub-domain nodes, fun for function

nodes, *sp* for service provider nodes, *service* for service nodes, *contract* for *Contract* nodes and *context* for *Context* nodes.

A role is represented as a tag attached to each node in Figure 23. *Roles* is a list of names that represent roles of service providers and requesters. These roles include two main lists. One is *RequesterRoles* another is *ProviderRoles*. The *RequesterRoles*are names of roles that are defined to control access for service requester. Every time SR starts a session with CASR, CASR assigns a role for SR. The role is decided by a third party of the system called Trusted Authority Unit (TAU). Assuming this TAU exists, when the role is assigned for SR, the domain, sub-domain or function is not displayed for SR unless the role of SR exists in *RequesterRoles* list of this particular node. Similarly, the *ProviderRoles* are also names of roles that are used to control access. However, it is specified to control access for service providers SP when they are publishing a service. The reason for not merging these two roles list together is because using a service is different from providing a service. Some services could be used by a wide range of roles or types of people with different professional skills or educational levels. Nevertheless, services can be provided by a limited range of roles that are assigned to few service providers with concern only on their legibility for providing such a service. For example, medical services could be used by almost every adult without specifications to their job or level of knowledge. However, a medical service is allowed to be provided by licensed doctors or clinics. Therefore, the two roles list are separated from each other.

The *ProviderRoles* list is defined by a third party that is intelligent to specify the roles of providing some *ConfiguredServices*. However, the *RequesterRoles* are initially born with the grandchild roles list(function). When a new role is added to the grandchild role list, an update operation is required to child (sub-domain) and parent role list (domain). This is because, SP is able to update *RequesterRoles* list every time service provider SP adds a new service. However, logically if the function is not allowed to be used for some specific role, parent and grandparent of this function should be aware of that. So, if the roles of a function are represented in a set

called *A*, and the roles of roles of its parent sub-domain are represented in a set called *B*, and the roles of its parent domain are represented in a set called *C*, then $A \subseteq B \subseteq C$ must hold.

Followers and *Followings* are concepts inspired from the social website *Twitter*. These two elements are used to build the relationship among users of *Twitter*. Similarly, *Followers* and *Followings* are used to build relationships among nodes of the CASR. *Followers* is a list that includes all the IDs of other nodes that follow this node. That is, *Followers* element of function, sub-domain, and domain nodes are to define respectively what providers are followers to a specific function, what functions are followers to a sub-domain and what sub-domains are followers to a specific domain. However, *Followings* is a single value that represents one ID that represents the parent of this node. So, *Followings* element of function and sub-domain are respectively define which sub-domain is the parent of this function and which domain is the parent of this sub-domain. The domain does not include following information as it is a root node that follows no node.

The first three levels of the hierarchy represent the *Domain Knowledge* (DKN). DKN is used to provide service requesters SR knowledge about services stored in the registry. In searching service registries that exist in practise, the service requester usually seeks a service by entering key words. In response, a huge number of services might turn up. Some of those services may not be relevant to the user. However, in our design the registry uses DKN to direct a service requester to specific groups of services. That is, CASR displays available domains of services and guides the user to navigate through domains of interest. Then, when SR selects a specific domain, the registry displays sub-domains under this domain, and the navigation continues. This directs SR to service which that are categorized under this specific area. So, our structure has the following advantage:

- *It increases manageability and Enhance performance.* searching for a specific service is narrowed to particular services without the need to go through all services that exist in

the registry. For example, if the service requester SR looks for specific service name, by a single query, it could be known if the service is included under this domain or not. By having all the service names stored in the domain attached with their IDs, it is easier to recognize service ID. Hence, the system can simply retrieve data for this particular service. This manages the workload on the registry and hence enhances performance.

- *It controls accessibility.* Not every domain of knowledge could be accessible for every requester. For example, let us assume that under the domain HealthCare a service provider SP, who is not licensed, wants to publish a service. By enforcing Access Controls our system will disallow this operation. Similarly, a user who is not authorized to browse certain parts of the registry will be forbidden to navigate through those sections of registry.

- *Syntax and semantics of representations help categorize services.* Service providers can publish similar services under two different categories. These two categories may be semantically the same but syntactically different. For example, the domain Entertainment and Games are syntactically different but semantically the same. Providing the Domain knowledge, with description to types of services that belong to a domain, service providers can choose the appropriate domain under which they want to publish a service.

5.1.2 Modeling Domain, Sub-Domain and Function

In this section, the definition of each node is explained thoroughly. These three nodes, as shown in Figure 24 have precise definitions in the system. They are to be used every time a domain, sub-domain or a function is to be constructed by the system or requested to be constructed by a service provider.

1. **Domain**

   The domain definition includes Name, *ProviderRoles*, *RequesterRoles*, *Level*, *NodeType*, *ChildType*, *Followers*, *SPContext*and *SRContext*. We have explained already *ProviderRoles*,

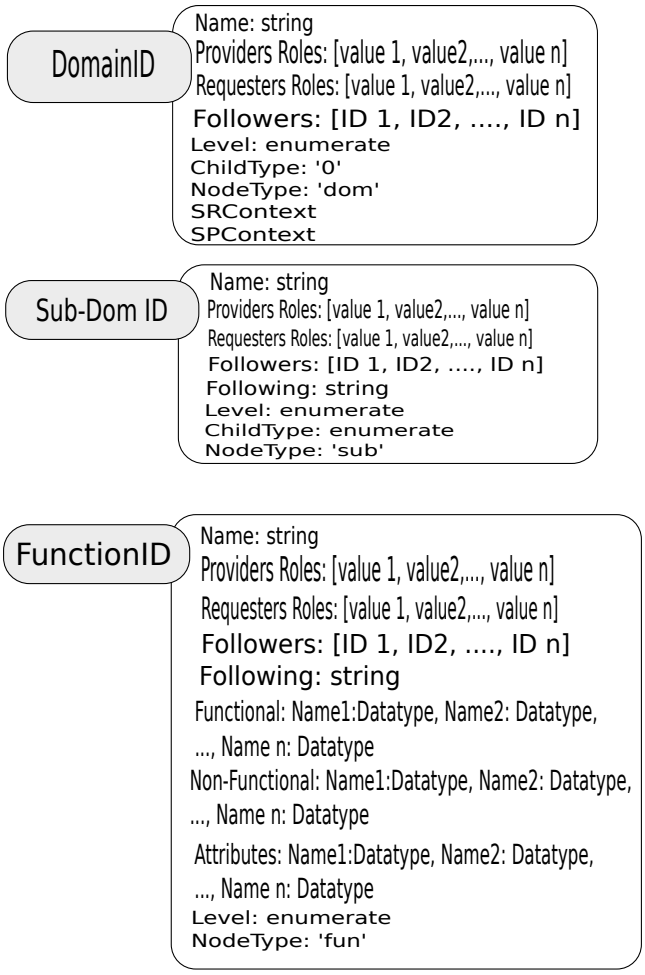


Figure 24: The Domain, Sub-Domain and Function Definition

RequesterRoles, and *Followers* in previous paragraphs. From modeling perspective, they need to include an identifier which is a key(unique name) that maps to a list of string values. The values of *RequesterRoles* and *ProviderRoles* list role names for the service requesters or service providers. With roles are associated the eligibility to browse this domain. The values of *Followers* list are the IDs of sub-domains that follow this domain. However, to model Name, *Level*, *NodeType*, and *ChildType*, each field needs to be mapped to a single value, a string value for name and *NodeType* fields and an enumerate value for *Level* and *ChildType* fields.

SPContext is the context of service provider that is pre-defined for this domain in the

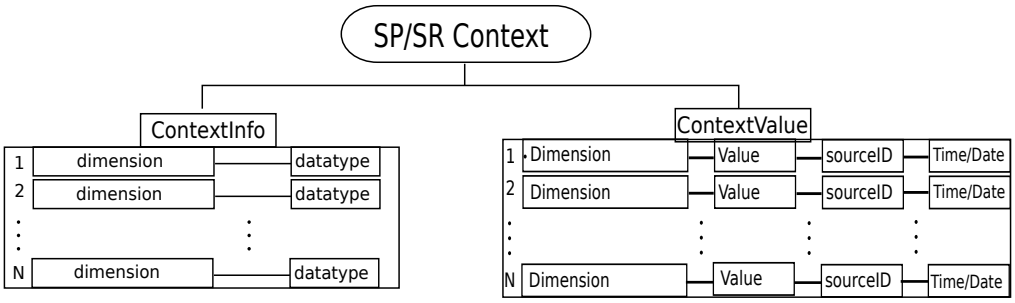


Figure 25: SP and SR Context Definition

system. The definition of context information and its values are represented in Figure 25. From Chapter 3 discussion it is clear that the structure of context can be used with any component. Therefore, we are using the same structuring, excluding information related to service restriction and context history. The *contextRule* is the field that includes the situation that the service is allowed to be executed in. The *SPContext* and *SRContext* does not require such an information. Thus, it is skipped. Also, the type of *SPContext* is *transient*, which means there is no need to store it in history. Also, there is no need to add registration information in the context. The *SPContext* and *SRContext*, which are the contexts of service provider and service requester respectively, are defined within a structure that is represented in Figure 25.

2. **Sub-Domain**

   The sub-domain node includes ID, *Level*, *NodeType*, *ChildType*, *ProviderRoles*, *RequesterRoles*, *Followers* and *Followings*. *Level*, *NodeType*, *ChildType*, and *Followings* the modeling principle for domain, a sub-domain is modeled. The *ProviderRoles*, *RequesterRoles*, and *Followers* are modeled as keys mapping to a list of string values.

3. **Function**

   The function node includes ID, *Level*, *NodeType*, *ChildType*, *ProviderRoles*, *RequesterRoles*, *Followers*, *Followings*, *Functional*, *Non-functional* and *Attributes*. *ProviderRoles*, *RequesterRoles*, *Followers*, and *Followings* are similar to their models in domain and sub-domain.

Figure 26: Provider information displayed with its service

However, *Functional*, *Non-Functional*and *Attributes* are the information stored in the service component of *ConfiguredService*. We model them as lists of keys associated to their data types. The data type is String. Figure 24 shows the general definition of function.

5.1.3 Service Providers (SP)

At this level we include information on service providers who provide a service with this functionality. This information is repeated with every function that they provide in a service. That is, if a service provider has services under different function nodes, then their information is repeated under each function. However, it is not repeated if the provider is providing more than one service under the same function. The provider information included in this node is part of the information that is stored in their account and profile. What is stored in this node is the information that matters to service requester such as name, rating and location. However, personal information of service providers are kept private in the accounts database. A service provider model includes the following information:

- ID
- Name
- Location
- Contact information
- Service Rating

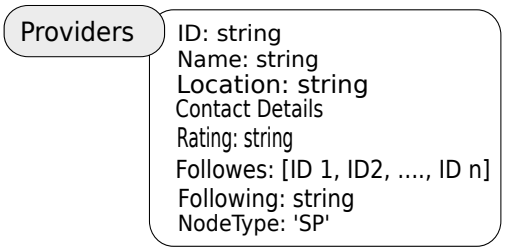


Figure 27: Provider Definition in the System

- *NodeType*

To capture this model we use a list of keys associated to their types. Figure 27 shows a generic model. The reviews and recommendation of a service is stored with the provider of that service as part of the *ConfiguredService*. The Service Rating information is repeated with every service because it is viewed by service requesters as they browse the registry. As an example, let us assume that a service provider with the profile

$$Id : Kje123, Name : JamesClow, Location : London, ON,$$

$$Email : JamesClow@hotmail.com, Rating : 4/5$$

provides an online service. While browsing such an online services, the service provider information is displayed as in Figure 26.

5.1.4 Configured Services (CS)

In Figure 23, the *ConfiguredService* part encompasses three levels of the hierarchy which are *Service*, *Contract*, and *Context*. The service as it is introduced in Chapter 4, includes the *Functional*, *Non-Functional*, and *Attributes* values of the service. Adding to it *NodeType*, *Followers*, and *Followings*. The *NodeType* is to define the type of this node. *Followers* is to list contracts belong to this service and *Followings* is to specify the ID of node's parent. These fields are modeled as they modeled with other nodes of the hierarchy. Previously, it is clarified that the contract includes *Context*, *Trustworthiness*, and *LegalRules* issues. The *NodeType*, *Followers*, and

Followings need to be added to contract elements. *NodeType* and *Followers* is modeled and defined the same. However, *Followings* is to list the IDs of contract parents. That is, it is not a single value as it is with other nodes. The relationship between a service and a contract is many to many. Hence, it is needed to link between those nodes from both ways. That is, from a contract node, the system is able to get all services that are parent to this node and from service node, the system is able to list all contracts belong to this node. *Context* is also defined and structured in Chapter 3. Adding to the elements included in context component, we add *NodeType* and *Followings*. The *NodeType* is to sign that this node is a context and *Followings* is to specify the contract parent of this node. The service can be related to many contracts and the contract can be related to many service. The contract links to one context only. Hence, One service with one contract attached to one context creates a unique *ConfiguredService*.

5.2 Context-Aware Service Registry Generic Model

With every component we discussed in previous chapters, we have provided a generic structure for implementation. In this chapter we discuss a generic model for the registry tree structure. This model includes mapping information among entities including *ConfiguredService*. Figure 28 illustrates the generic model of registries tree structure. It shows that every domain can be a parent of one or more sub-domain(s) which can be a parent of one or more function(s). A function is then mapped to one or more service provider(s). For the sake of simplicity, the provider entity is created under every function which is the functionality of a service provided by the service provider. If the service provider providers several services under the same function, the entity is not duplicated but mapped to several services as shown in Figure 28.

Example 13 *Lets, assume that we have domain, sub-domain, function and provider that are defined respectively as follows:*

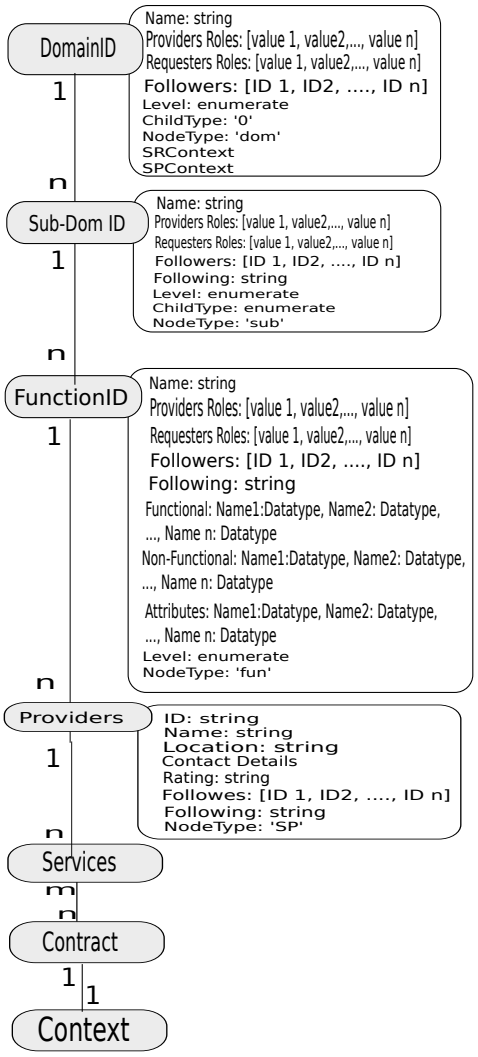


Figure 28: The Generic Structure of Service Registry Domain Knowledge and Providers

$$\textbf{\textit{Dom1}} \rightarrow [(Name : Transportation), (Level : 0), (NodeType : dom), (ChildType : 0),$$
$$(ProvidersRoles : [r1, r2, r5]), (RequesterRoles : [r12, r13]), (Followers : [sub1, sub2])$$

$$\textbf{\textit{sub1}} \rightarrow [(Name : CarServices), (Level : 1), (NodeType : sub), (ChildType : 1),$$
$$(ProvidersRoles : [r1, r5]), (RequesterRoles : [r12]), (Followers : [fun1]),$$
$$(Following : Transportation)]$$

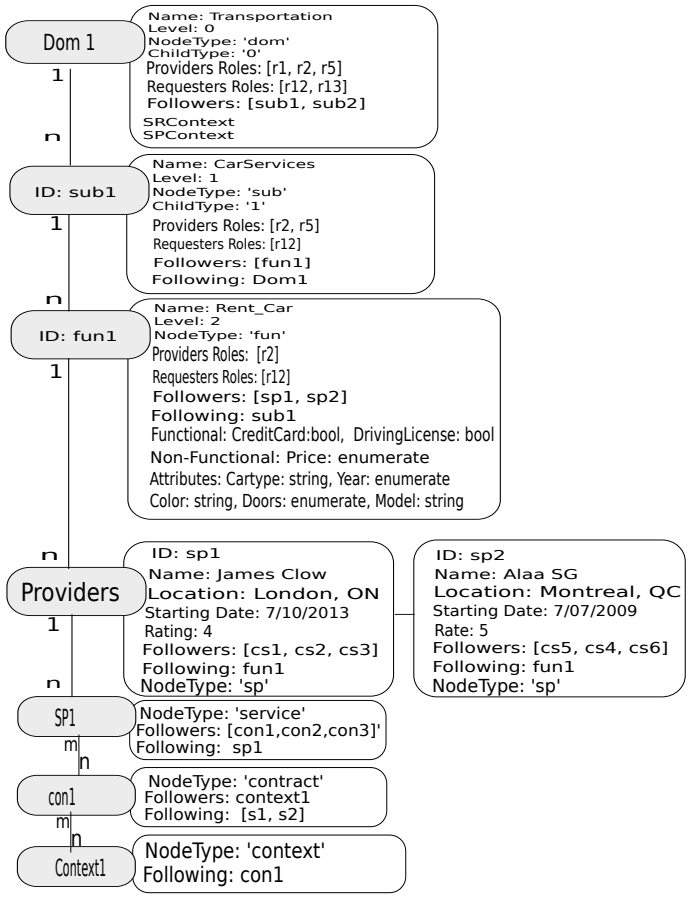


Figure 29: Example of Tree Structure for Transportation Domain

$$\textbf{\textit{fun1}} \rightarrow [(Name : Rent_Car), Level : 2, NodeType : fun,$$
$$(ProviderRoles : [r1]), (RequesterRole : [r12]), (Followers : [sp1, sp2]),$$
$$(Following : sub1), (functional : [(creditCard : bool), (drivinglicense : bool),$$
$$(nonfunctional : [(price, enumerate)]), (attributes : [(carType : string),$$
$$(Year, enumerate), (color, string), (doors, enumerate), (model, string)])]$$

$$\textbf{\textit{sp1}} \rightarrow [(Name : JamesClow), (Location : London, ON), (Email : JamesClow@hotmail.com),$$
$$(Rate : 4/5), (Followers : [cs1, cs2, cs3]), (Following : fun1, (NodeType : sp)]$$
$$((\textbf{\textit{sp2}} \rightarrow [(Name : AlaaSG), (Location : Montreal, QC), (Email : alaa - sg@hotmail.com),$$
$$(Rating : 5/5), (NodeType : sp), (Followers : [s3]), (Following : fun1)]$$

$\boldsymbol{s1} \rightarrow [(Name : service1), (Followers : [con1, con2]), (Following : sp1, (NodeType : service)]$

$((\boldsymbol{con1} \rightarrow [(NodeType : contract), (Followers : [context1]), (Following : s1, s2)]$

$(\boldsymbol{context1} \rightarrow [(NodeType : context), (Following : con1)]$

This data is represented in Figure 29.

■

5.3 Summary

In this chapter, the context-aware service registry CASR storage is discussed covering structure information that is used to classify the massive row data of *ConfiguredServices*. The CASR structure supports hight manageability and controlled accessibility by providing and storing role information. After structuring the registry, we are going to discuss its implementation. The implementation needs a database that structure data and stores them. In the following Chapter, database selection, design and implementation are discussed.

Chapter 6

CASR Implementation in NoSQL Databases

In this chapter, we customize the general structure introduced in the previous chapter to the three NoSql databases Redis, MongoDB, and Hbase. First, we introduce a general description of NoSql databases and explain why we preferred them over traditional databases. Then, we narrow down the discussion on the three NoSql databases. There are various options to implement each NoSql database type. Therefore, we decided to pick only one representative implementation for each type. For each database, we present general technical information and then use this information to map the general structure provided in Chapter 5 to a structure that suits the characteristics of each database.

6.1 Why NoSql Databases

The decision to select a database platform for a specific system is dependant on the data that is required to be stored in a database. When using a database to organize unsupported structure of data, the potential of the target database should be fully achieved. This is why we decided

using NoSql databases. Given the characteristics of our data, which was described in the previous chapters, traditional relational databases (RDBs) are not the best option. In fact, RDBs were created to host and manage similar data which are grouped in tables. It support schema-based structures. Also, RDBs have fixed number of columns for every table. This means that rows in a table should have the same number of fields [Lea10]. Therefore, data included in one table cannot be heterogeneous in the sense that they all should have the same number of fields. This causes an increase in the number of tables and in turn increases the number of join operations to link those tables. Join operations are very costly, CPU intensive and memory consuming. However, because of the rigorous pre-fixed structure of columns in a table, joining operations are unavoidable in RDBs. Although many efforts have been put [Cat11] towards enhancing the scalability and the efficiency of RDBs by using *master-slave* and *sharding* techniques, they fall short when faced with high volumes of diverse structures of data. Distributing RDBs is complex and storing data in one machine might require expensive maintenance operations when the data load increases. Having the data stored in one place with pre-fixed schema help providing safe transactions featured with Atomicity, Consistency, Isolation and Durability (ACID) [Cat11]. SQL-like databases have pre-fixed structured schema which provides efficient query system and aggregation operations. CASR registry data adopts highly diverse structures. It contains different data structures and numerous levels of hierarchy. Service-Oriented Architectures are mainly built to respond to clients efficiently. Also, they are meant to store massive data that are not necessarily structured. Therefore, RDBs is not the best choice to implement the context-aware service registry.

NoSQL can fully contain the data of CASR registry. NoSql databases are naturally distributed. They support free-schema structures, diverse structure of rows and data hierarchy. Therefore, our data can be denormalized into less number of tables which can eliminate the great cost associated with join operations. Also, NoSql databases can store high volume of data. Many experiments and comparative studies have been conducted on the measurable difference

in performance between NoSql and RDBs [Cat11]. NoSql does not support ACID transactions. However, they support Scalability, Consistency, Availability and Partition Tolerance. NoSql can query data; however, in general, they do not support aggregation operations such as `Group By`. The simplicity of NoSql database structures and its distributed nature reduces the need for management.

The main reasons for choosing NoSQL over RDBs model to implement CSAR are:

- NoSql can handle heterogeneity, withstand a large number of read and write operations, and avoid join operations. Consequently, there is likely to be less workload and better performance.
- NoSQL databases support massive database management efficiently.
- NoSql technology supports semi or free schema which makes it suitable for managing semi-structured data [TJ10].

In service-oriented systems we need to provide flexible service schema for each service provider. That is, not all service providers can be forced to stick to one particular service schema. Every service provider should be given the flexibility to choose a schema that best fits their service description goals. It is important to emphasize that the CSAR structure provides a generic service model, which makes it possible to model any service in it. Consequently we aim to provide its implementation in a database model which does not restrict the service schema to be identical for all service providers.

There are different types and options when it comes to NoSql databases. However, they mainly fall under the following three categories: 1) *Key-Value*, 2) *Document-Oriented*, and 3) *Column-Oriented*. Each of these NoSql technologies has many platforms to support its operations. Therefore, we decided to choose one NoSql platform from each category in order to implement CASR. The selected databases are: 1) *Redis* for Key-Value, 2) *MongoDB* for Document-Oriented, and 3) *Hbase* for Column-oriented.

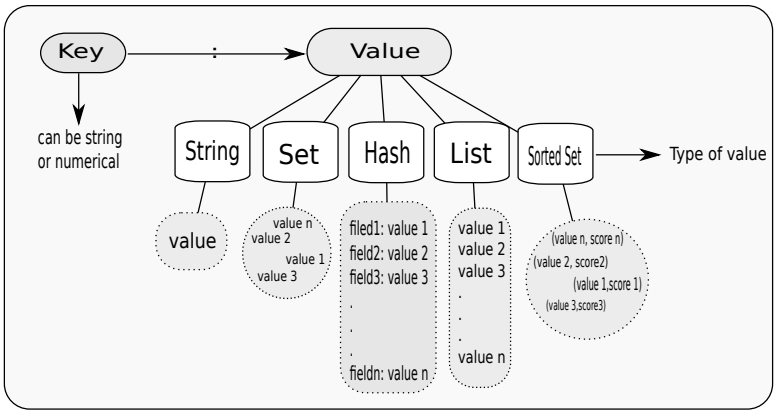


Figure 30: Redis Key-Value Store

6.2 Implementation: Redis - Key-Value Store

Redis is an open source advanced *key-value* database. In this section we describe in some detail the features of Redis shown in Figure 30.

6.2.1 Redis Features

A record in a key-value database consists of a key mapped to its corresponding value. Redis is considered an advanced key-value database because it provides five possible data structures for the value type. These data structures are explained below.

- *String*: A String type is suitable to store a single value with a maximum size of 512MB. The value of a string can be any byte array such as an integer counter, a string or a binary serialized object. String structure is supported with different commands and operations [SN10]. However, the CRUD operations GET, SET, and DEL are the most used ones. Updates and insertions are performed by SET, retrieval of information is done by GET and DEL is used to delete a record. These are explained in the screen shot shown in Figure 31.

- *Hash*: A Hash type is suitable to store a set of pairs where each pair consists of a name (field) and its corresponding value. A single Hash record in Redis could have up to $2^{32}-1$

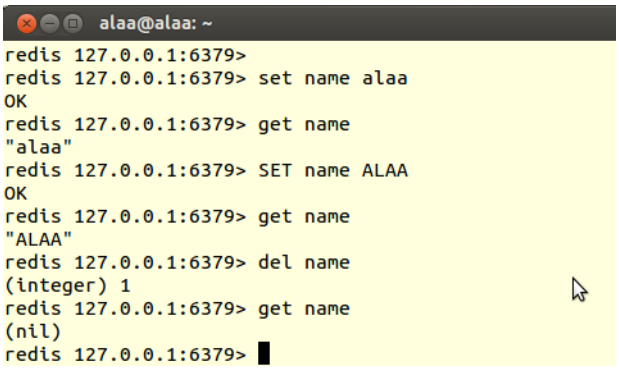


Figure 31: String Commands in Redis

pairs. Hash is similar to a hash directory which allows one to manipulate data through the fields. Redis provides operations such as $HSET$ and $HMSET$ to set single field and multiple fields at a time. Retrieval from hash storage is done using $HGET$ operation with $keyfield-Name$ as parameter. The operation $HGETALL$ is used to retrieve all the fields and their values. Redis system is intelligent to recognize all existing fields and update them with the new data or add the new entered field to the assigned key. Finally deleting the whole hash record can be performed using either the command used with string type or $HDEL$ command. Figure 32 shows a a Redis session in which hash commands are used.

- *Set*: A Set is suitable to store an unsorted and not duplicated group of elements connected to a single key. In a Set data structure of Redis, a maximum of $2^{32}-1$ elements can be stored. For Set data structure the operations $SADD$, $SREM$, $SMEMBER$, $ISMEMBER$, $SUNION$, and $SINTER$ are provided in Redis. Their semantics follow the set theory semantics for inserting, deleting, checking membership, and forming respectively the union and intersection. For instance, the operation $SADD$ will need an element and set variable as input. If the input element is not in the input set, the element is inserted in the set, otherwise the input set is remained with no change. There is no operation for updating the members values in a set, however it can be performed by deleting the

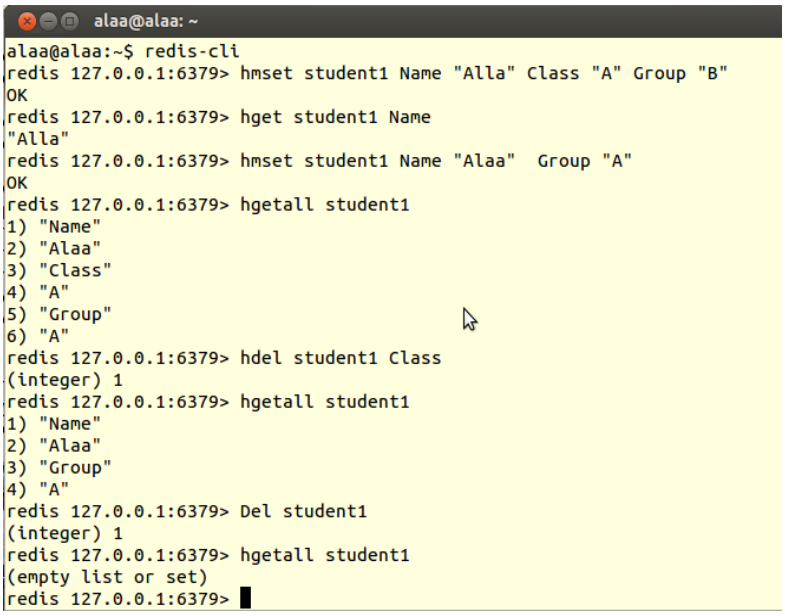


Figure 32: Hash Commands in Redis

appropriate member from the set and then performing the add operation. In Figure 33 we show a sample Redis session with set operations.

- *List*: A List type is simply a list of string values that are ordered as they are entered. A List data structure of Redis could have a maximum size of $2^{32} - 1$ values. List uses the command $LPUSH$, $LPOP$ to add or remove a value to the list. List type provides the operation $LLEN$ to determine the length of a list, the operation $LSET$ (with $key, index$ as argument) to index the elements of the list, and the operation $LRANGE$ (with arguments $(keystart, keystop)$) to retrieve a range of values. Figure 34 shows a sample session with List operations in Redix.

- *Sorted Set*: A Sorted Set type is a Set type, in which each value is associated with a score. A score is an integer number attached to each value of a Sorted set. The values of a Sorted set are sorted in ascending order based on the scores. To add a member to a sorted set the command $ZADD$ with arguments key and $score$ is used. The command $ZCARD$ with argument key is used to get the member of a specific sorted set. The operation

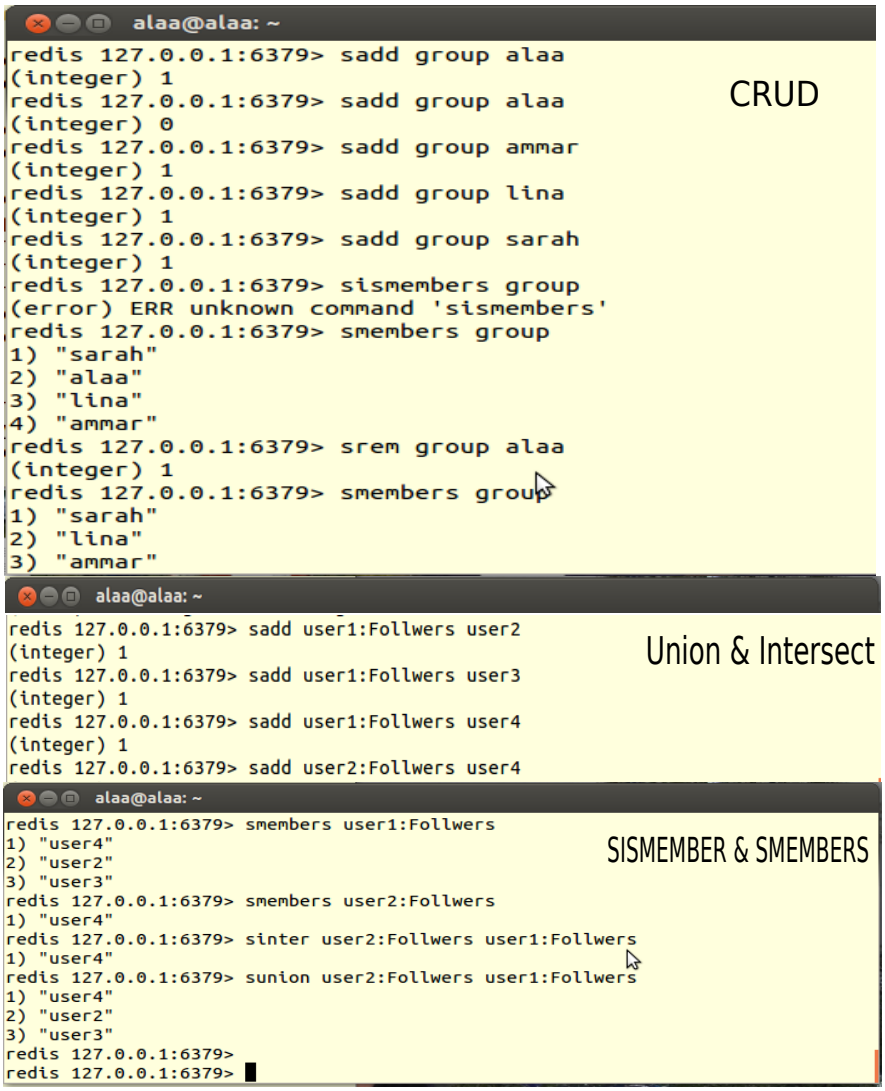


Figure 33: Set Commands in Redis

ZRANGEBYSCORE with arguments $keyfrom$ and $keyto$ is used to get the values of the members from the index $keyfrom$ to $keyto$. Updating a sorted set is similar to updating a set [SN10] [RWC12]. Figure 35 shows a Redis session using Sorted Set structure.

6.2.2 Design Considerations

Mapping the generic model of CASR, introduced in Chapter 5, is not trivial. This is because the nature of key-value stores does not allow constructing one rich object wrapped in a table.

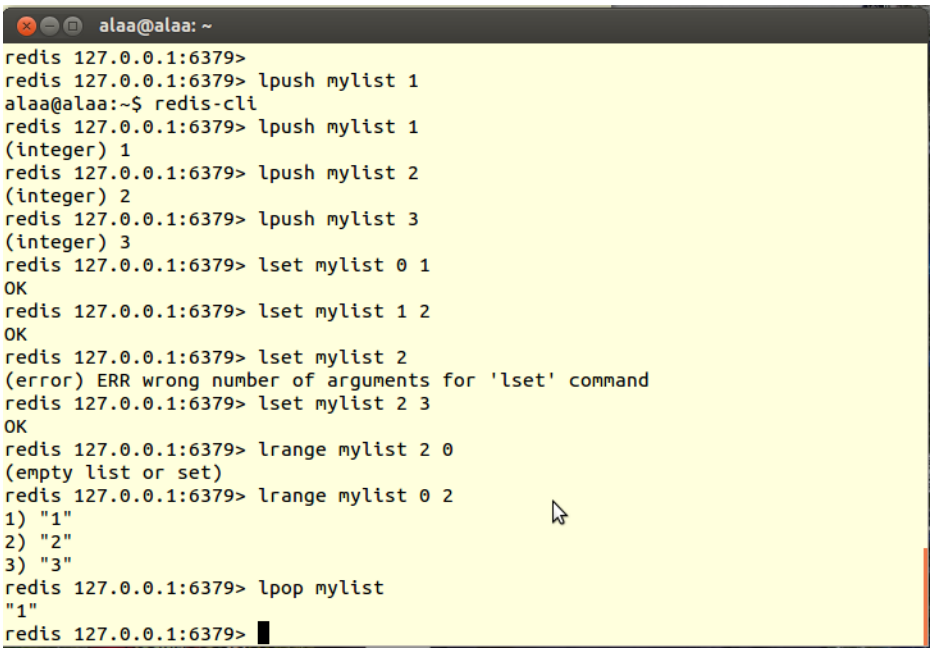


Figure 34: List Commands in Redis

Figure 35: SortedSet Commands in Redis

Therefore, the following design issues should be resolved first.

- It is necessary to aggregate data together to construct conceptual tables. Hence, we link records of the same conceptual tables in a clear way. There are two solutions for wrapping related data to a single record, as explained below:

  1. Key Pattern: We can construct a key, by combining keys along the path from the

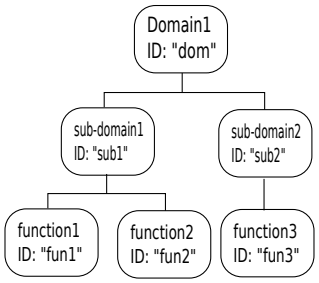


Figure 36: Key Pattern Example

root to a child node. This is explained in the following example.

Example 14 *In Figure 36, domain is the root of the hierarchy and its key is* dom*. The two sub-domains* $sub-domain1$ *and* $sub-domain2$ *are the children of the root, and their respective keys are* $sub1$ *and* $sub2$*.* $Sub-domain1$ *has the two children* $function1$ *and* $function2$*, whose keys are* $fun1$ *and* $fun2$*.* $Sub-Domain2$ *has one child* $function3$ *whose key is* $fun3$*. In Redis, as duplicating keys is not allowed, we cannot represent*

$$dom : \text{“}sub1\text{”}$$

$$dom : \text{“}sub2\text{”}$$

So we construct the pattern $dom : sub1$ *as the key for* $sub-domain1$*, and* $dom : sub2$ *as the key for* $sub-domain2$ *and represent them in Redis as follows:*

$$dom : sub1 : \text{“}sub : domain1\text{”}$$

The three functions are represented in Redis as shown below:

$$dom : sub1 : fun1 : \text{”}fun1\text{”}$$

$$dom : sub1 : fun2 : \text{”}fun2\text{”}$$

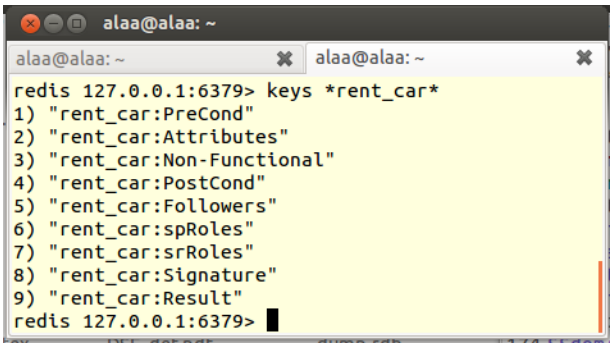


Figure 37: Retrieving all records by patterns

This method produces long complex IDs. Also, we do not need to know the path from keys as we included *Followings* and *Followers* information defined in the generic model. Hence, we thought of another pattern which is

$$NodeID : FieldName : \ value$$

If this pattern is used, it is possible to recognize the node's parameter represented by this record. For example, in *ConfiguredService*, the service part is represented by four elements which, *functional*, *Non-Functional*, *Attributes*, *Signature*, AND *Result*. Each of these is represented in a separate record in Redis. Their keys are represented as $FuncID : Attribute$, which helps us to distinguish elements from one function to another. So, using the key patten command, we can get all the elements related to one service. The key pattern command helps to find all the records that match a pattern. This command is $keys**$. In Figure 37 we shown a sample Redis session in which the pattern $*rent\_car*$ is used as an ID of $rent\_car$ function. We found that this is very useful for retrieving data related to a specific node. Therefore, we applied it in our structure.

- Data Type Selection: We should choose the appropriate Redis data type by examining the nature of included data and query types on this information.

- Secondary key: Redis does not support secondary keys. If we want to allow a user to

search the database using a secondary key such as email or phone number, then we can use the Redis features that allow searching on the value of any given attribute. That is, we can first look up for matches based on email and then for services that match for number and combine result. Alternatively, we can create additional mappings from the generic model to the Redis database. From the information stored in the generic model, we can identify the primary key and secondary keys for each model element. For each secondary key, we can construct the map which maps it to the primary key. For example, the map $Email \rightarrow ID$ maps all emails to the respective IDs. We can store this key value pairs in Redis and use it when the user requests a search based on email. Consequently, we have to store all possible maps of secondary key to the respective primary keys and store them in Redis data base in order to allow the user to search the data based on secondary key. This will create additional storage and a small overhead of searching time for the primary key.

6.2.3 Implementing the Domain Knowledge Design

To map the DKN that includes Domain, sub-domain and function in the generic structure into Redis, we decided to construct the tables 1.1 to represent domains, 1.2 to represent sub-domains, and 1.3 to represent functions.

Table 1.1

Domain:ID (name:value, ID:value, SPContext:ID, SRContext:ID, Level:value, childType: value, NodeType: value) (hash)

Domian:ProviderRoles (set)

Domian:RequesterRoles (set)

SPContextDefintion: (name:type, name:type, name:type) (hash)

SPContextValue: (name:value, name:value, name:value) (hash)

SRContextDefintion: (name:type, name:type, name:type) (hash)

SRContextValue: (name:value, name:value, name:value) (hash)

Table 1.2

SubDomianID:Info (Name:value, Level:value, Following:value, childType:value, NodeType: value) (hash)

SubDomianID:ProviderRoles (set)

SubDomianID:RequesterRoles (set)

SubDomainID:Followers (set)

Table 1.3

FunctionID:Info (Name:value, Level:value, Following:value, NodeType: value) (hash)

FunctionID:RequesterRoles (set)

FunctionID:ProviderRoles (set)

FunctionID:CSList (set)

FunctionID:Followers (set)

FunctionID:PreCond (name:type, name:type, name:type) (hash)

FunctionID:PostCond (name:type, name:type, name:type) (hash)

FunctionID:Attrib (name:type, name:type, name:type) (hash)

FunctionID:Sign (name:type, name:type, name:type) (hash)

FunctionID:Result (name:type, name:type, name:type) (hash)

FunctionID:NonFunc (name1:type, name1_des:value, ... name n:type, name_des n:value)

To structure DKN in Redis we use the data types *Set*, and *Hash*. For implementing the fields *Followers*, *ProviderRoles*, and *RequesterRoles* we use *Set* type. This is because these fields store list of their children IDs. Thus, *Set* is a proper data type to store these unrepeated set of values. With sets, we are able to retrieve all elements at once, check existence of a specific value, and delete/add a single member of/to the set. However we represent *Attributes*, *Functional*,

and *Non-Functional*in *Hash* data type. This is because it provides the ability to include data in $key : value$ format. From the generic model representation *Attributes*, *Functional*, and *Non-Functional*, it is clear that each one of these elements needs a list of values associated with their names. In Redis this representation is made possible with hash data structure. The rest of the information including Name, *Level*, *NodeType*, *ChildType*, and *Followings*, which are parameters with a single value, can be represented in a separate record using *String* data type. However, we decided to have them all stored in one *Hash value* that wraps all fields that have single values. That is, instead of storing the information $Name$, *Level*, *NodeType*, *ChildType*, and *Followings* of a sub-domain $sub1$ in five different records,

$$sub1 : Name : \text{“}Sub - domain1\text{”}$$

$$sub1 : NodeType : \text{“}sub\text{”}$$

$$sub1 : Level : \text{“}1\text{”}$$

$$sub1 : ChildType : \text{“}1\text{”}$$

$$sub1 : Following : \text{“}dom\text{”}$$

we use the hash data type to store all the above information in one record, named $Info$, so $sub1 : info =$

$$[Name : \text{“}Sub - domain1\text{”} NodeType : \text{“}sub\text{”} Level : \text{“}1\text{”} ChildType : \text{“}1 Following : \text{“}dom\text{”}]$$

For Domains, the record $Info$ includes IDs of *SPContext*, and *SRContext* definition and values. These two fields are the “LinkingTo” fields. That is, they include IDs of other records related to this domain. Each *SPContext* (*SRContext*) is represented by two *Hash* records. One for defining the dimension (names) with their data types and the other for associating those dimensions with their values.

The sub-domain and domain have similar structure. In table 1.2, we use *hash* type to wrap together $Name$, *Followings*, *NodeType*, *ChildType*, and *Level* fields. We use Redis *Set* type

to represent *Followers*, CSList which includes grandchildren(services) IDs, *ProviderRoles* and *RequesterRoles*.

In table 1.3, a function definition includes $Info$ record to map the defined fields in the generic model. However, from the generic model function node includes *Functional*, *Non-Functional*, *Attributes*, *Signature*and *Result* definitions. Each one of them must be represented individually. Such an individual representation requires the key pattern we defined earlier to link the records of one function together. That is, FunctionID:Attributes is the key for Attributes record of function1 that is represented by including its ID before the colon of this record key. This way, using the $Key**$ command of Redis, we are able to get all the records of a function. For example writing the command Key "$FunctionID$", the records keys FunctionID:Info, FunctionID:RequesterRoles (set), FunctionID:ProviderRoles (set), FunctionID:CSList, FunctionID:Followers, FunctionID:PreCond, FunctionID:PostCond, FunctionID:Attrib, FunctionID:Sign, FunctionID:Result, FunctionID:NonFunc are all retrieved. Hence, we are able to get their information by using a query with a key.

For the function node, we include the list of attributes and their type but not the values of the attributes. This is because at the function node, we include the attributes of all the *ConfiguredServices* under that node. The value of the attributes along with their types are listed for each *ConfiguredService* which the grandchildren of the function node. Such a representation will enable all the *ConfiguredServices* that can be browsed for a given set of attributes. At the time of publishing a *ConfiguredService*, the system checks the fields of each part and will incrementally add descriptions. The following example illustrates this incremental representation.

Example 15 *Let CS1 be a* ConfiguredService*, which is a grand child of function (fun1), in which the* $Attribute$ *parameter of CS1 which is described by two lists.*

$$AttrDef : [a_1 : type_1, a2 : type_2, a3 : type_3],$$

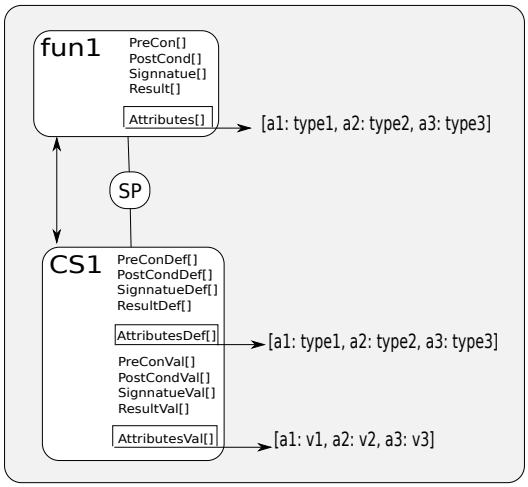


Figure 38: CS1 published as a grandchild of fun1

and

$$AttrVal : [a_1 : v_1, a_2 : v_2, a_3 : v_3],$$

The fields a_1, a_2, a_3 *are in both lists. Their values* v_1,v_2, v_3 *are stored in AttrVal. The type of* v_i *is* $type_i$, *and is stored in* $AttrDef$. *That is* $AttrDef$ *is a definition to* $AttribVal$. *Note that the hash type in Redis, see table (1.4)(1.3), allows us to have a list of attributes.*

When CS1 is published by the system it adds the definition of attribute to fun1 node which is represented as follows:

$$Attributes : [a_1 : type_1, a_2 : type_2, a_3 : type_3]$$

Note that, the value parts are added under the service Figure 38. Assuming that another ConfiguredService *, CS2, is added with the following* Attributes *lists representation:*

$$AttrDef : [a_1 : type_1, a_4 : type_4],$$

and

$$AttrVal : [a_1 : v_1, a_4 : v_4]$$

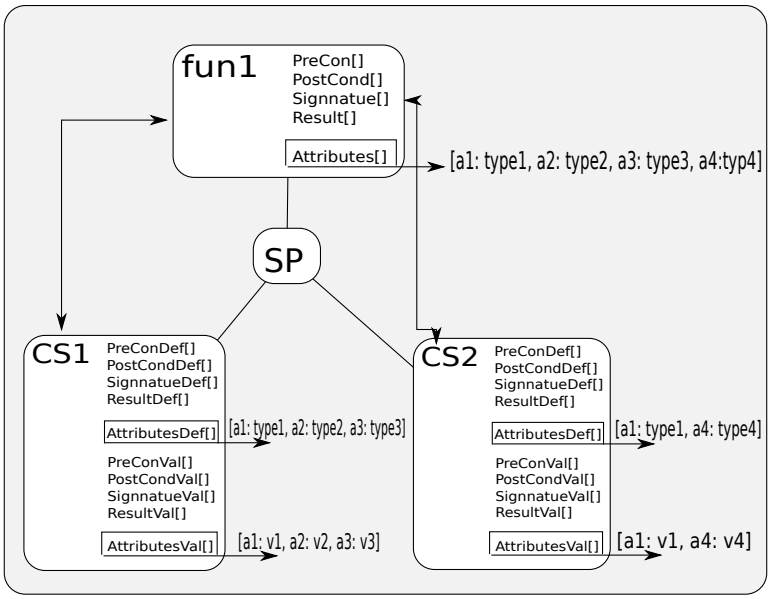


Figure 39: CS2 published as a grandchild of fun1

When CS2 is published, the $fun1$ attribute elements are simply updated as:

$$Attributes: [a_1 : type_1, a_2 : type_2, a_3 : type_3, a_4 : type_4]$$

Redis conceptual representation showing the incremental additional function description are shown in Figure 38 and Figure 39.

■

The inclusion of elements definition within the function parent entity has two advantages. The first advantage is to give a SP information on the attributes that already exist in the CASR when it is publishing the service. The SP can see the provided fields in similar services for each component and then they can be used to describe the service. Also, by including attributes definition in function nodes, the system is able to display all the services under this node. That is, when SR browses the available domains, sub-domains and functions, SR can select the desired function. When the function is selected, the service registry provides the defined attributes under this particular function; hence, it provides the service characteristics which the

SR is looking for. Consequently, the list of available services presented to SR will include all relevant services related to the SR requirement.

6.2.4 Implementing Provider Design

Service provider SP information is different from the information provided as part of the DKN as well as from the information included in the *ConfiguredServices*. For conceptual clarity, we separate service provider information from the rest. We map the service provider information in the generic model to a table structure. This table (1.4) includes one *Hash* data type that includes SP $Info$ which has the $Name$, $Address$, $Rating$, $Following$ and $NodeType$ fields. It also uses a *Set* data type to represent *Followers* field that includes all IDs of *ConfiguredServices* provided by the SP.

Table 1.4

ProviderID:Info (Name:value, Location: value, Rating:value, Following:value, NodeType:value) (Hash)

ProviderID:Followers (set of CS IDs)

6.2.5 Implementing Configured Service Design

We map the *ConfiguredServices* model to Table 1.5 and Table 1.6.

Table 1.5

ServiceID:Info (Name:value, Following:value, NodeType:value)

ServiceID:PreCondDef (name:type, name:type, name:type) (hash)

ServiceID:PreCondVal (name:value, name:value, name:value) (hash)

ServiceID:PostCondDef(name:type, name:type, name:type) (hash)

ServiceID:PostCondVal(name:value, name:value, name:value) (hash)

ServiceID:NonFuncDef (name1:type, name1_des:value, ... name n:type, name_des n:value)

ServiceID:NonFuncVal (name1:value, name1_des:value, ... name n:value, name_des n:value)

ServiceID:AttribDef (name:type, name:type, name:type) (hash)

ServiceID:AttrinVal (name:value, name:value, name:value) (hash)

ServiceID:SignDef (name:type, name:type, name:type) (hash)

ServiceID:SignVal (name:value, name:value, name:value) (hash)

ServiceID:ResultDef (name:type, name:type, name:type) (hash)

ServiceID:ResultVal (name:value, name:value, name:value) (hash)

ServiceID: Followers (set)

In the generic model, there is an over lap of information between the function and *ConfiguredService*. In the generic model, a *ConfiguredService* is a child of a service provider node, who in turn is a child of a function node. The set of attributes at a function node is a superset of the set of attributes of the services under it. That is, at the level of service node the attributes and value information are those that are very specific to that service. Consequently, when we map a service into Redis we need two records for each of *Functional*, *Non-Functional*, *Attributes*, *Signature*and *Result*. In one record we map the fields corresponding to the attributes names and their data types. In the other record we map the fields names and their values. For each element, it is needed to know the type of data associated with each field as this helps the system to provide other functionalities that deal with these data such as validation and calculation. Figure 40 shows a simple interface to enable the user to create such table service publication stage. Once services are uploaded for publication, an implicit update operation to the grandparent node, which is a function, is invoked. This invocation is to add the definition of new attributes to the Attribute list at the function node.

Table 1.6

ContractID:Info (Name:value, NodeType:value)

ContractID: Followers (set)

ContractID: Followeing (set)

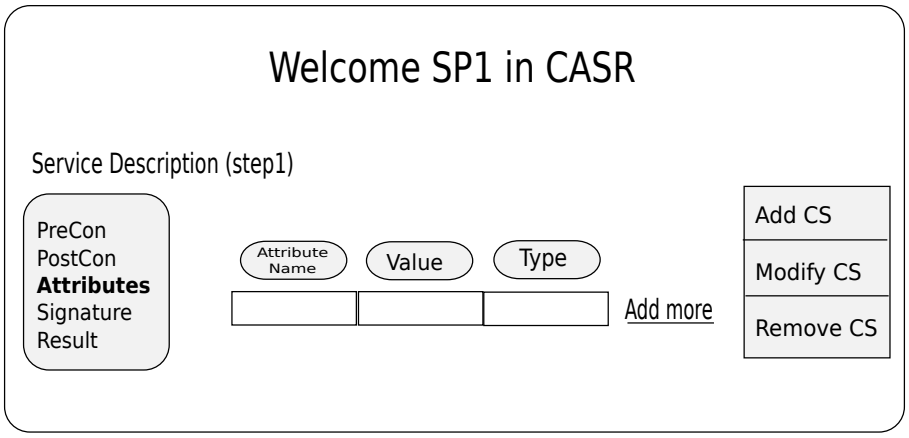


Figure 40: Basic Visualization for Service Publication Web Page

ContractID:LegalIssueInformal (name:value, name:value, name:value) (hash)

ContractID:LegalIssueFormal (name:value, name:value, name:value) (hash)

ContractID:ProviderTrustDef (name:type, name:type, name:type) (hash)

ContractID:ProviderTrustValue (name:value, name:value, name:value) (hash)

ContractID:ServiceTrustDef (name:type, name:type, name:type) (hash)

ContractID:ServiceTrustValues (name:value, name:value, name:value) (hash)

Table 1.6 shows the Redis table created by mapping the *Contract* part of the generic model. The contract node in the generic model includes definition record and values record for each of *ProviderTrust* and *ServiceTrust*. Both of these parameters need to be associated with type and value. However, unlike service node, the inclusion of type information does not lead to update operation in the function node. This is because the contract information is not included in the function node. Also, in *Contract* table, the *Followings* and *Followers* are defined with set Redis datatype. This is because the relationship between service part and *Contract* part are many to many, which means that *Followings* field includes several IDs for services they belong to. However, a *Contract* can be followed by only one *Context* although it includes several fields. Therefore, *Followers* field includes the contextID which can get the rest of fields ID such as

contextID:ContextInfo, contextID:ContextValue, and contextID:Rule.

LegalRules are part of *Contract* and they do not need a type field because it includes values represented as text which is a string value that is defined implicitly by the system. However, we need two records to map each legal rule, one to represent the formal representation and another to store the informal representation. That is, the informal value includes a list of fields which represent the rule name and its textual value. However, the formal representation of this rule is stored in the second record of *LegalRules* which is used and read by the system. The SP, who publishes the service, is responsible for entering the formal and informal representation of each rule assuming that SPs have the knowledge to do so.

6.2.6 Implementing Context Design

Table 1.7

ContextID:Info (name: value, NodeType:value (hash) ContextID:Following (set) ContextID:Rule (value) (string)

ContextID:contextInfo (Dim1:type, Dim2:type,..., Dim n:type) (hash)

ContextID:contextValue (Dim1:ID, Dim2:ID,..., Dim n:ID) (hash)

ContextID:Dim1ID (Dim1:value, Time:value, Date:value, SourceID:value) (hash)

ContextID:Dim2ID (Dim2:value, Time:value, Date:value, SourceID:value) (hash)

.

.

ContextID:DimnID (Dim n:value, Time:value, Date:value, SourceID:value) (hash)

Table 1.7 shows the Redis key-value model for context structure. The model uses String and Hash types to model the elements. The $Info$ record is added to include *Context* $Name$ and *NodeType* using hash Redis datatype. Because *ContextInfo* consists of many pairs of dimension names and their types, Hash is a good data type to use. Similarly, *ContextValue* contains pairs.

Figure 41: CASR Implementation in Redis (Snapshot(1))

However, it has two levels of hierarchy. The first level is used to map the names of dimensions to their value keys. The second level is used to map the value keys to nested Hashes that include dimensions information. The *ContextRule* attribute is modeled as a String data type because it contains only one value which is the *ContextRule* statement.

Table 1.4

```
redis 127.0.0.1:6379> hgetall sp1:info
 1) "name"
 2) "JamesClow"
 3) "rating"
 4) "4/5"
 5) "location"
 6) "Montreal,QC"
 7) "contact"
 8) "JC@CASR.com"
 9) "following"
10) "fun1"
11) "NodeType"
12) "sp"
redis 127.0.0.1:6379> smembers sp1:followers
1) "s1"
2) "s2"
```

Table 1.5

```
redis 127.0.0.1:6379> hgetall s1:info
1) "name"
2) "JamesCarRental"
3) "following"
4) "sp1"
5) "NodeType"
6) "service"
redis 127.0.0.1:6379> hgetall s1:PreCondDef
1) "validC"
2) "bool"
3) "validD"
4) "bool"
redis 127.0.0.1:6379> hgetall s1:PreCondVal
1) "validC"
2) "T"
3) "validD"
4) "T"
redis 127.0.0.1:6379> hgetall s1:PostCondDef
1) "confirmCredential"
2) "bool"
redis 127.0.0.1:6379> hgetall s1:PostCondVal
1) "confirmCredential"
2) "T"
redis 127.0.0.1:6379> hgetall s1:AttributesDef
 1) "carType"
 2) "string"
 3) "color"
 4) "string"
 5) "model"
 6) "string"
 7) "year"
 8) "int"
 9) "doors"
10) "int"
11) "transmission"
12) "string"
13) "companyName"
14) "string"
redis 127.0.0.1:6379> hgetall s1:AttributesVal
 1) "carType"
 2) "Toyota"
 3) "color"
 4) "red"
 5) "model"
 6) "echo"
 7) "year"
 8) "2013"
 9) "doors"
10) "4"
11) "transmission"
12) "automatic"
13) "companyName"
14) "CCCRental"
redis 127.0.0.1:6379> hgetall s1:NonFunctionalDef
1) "price"
2) "real"
3) "description"
4) "string"
redis 127.0.0.1:6379> hgetall s1:NonFunctionalVal
1) "price"
2) "50"
3) "description"
4) "us dollar"
redis 127.0.0.1:6379> hgetall s1:SignatureDef
1) "address"
2) "string"
redis 127.0.0.1:6379> hgetall s1:SignatureVal
1) "address"
2) "XXXX, Montreal QC"
redis 127.0.0.1:6379> hgetall s1:ResultDef
1) "bookingConfirem"
2) "bool"
redis 127.0.0.1:6379> hgetall s1:ResultVal
1) "bookingConfirem"
2) "T"
redis 127.0.0.1:6379> smembers s1:followers
1) "con2"
2) "con1"
```

Table 1.6

```
redis 127.0.0.1:6379> hgetall con1:info
 1) "name"
 2) "contract1"
 3) "NodeType"
 4) "contract"
redis 127.0.0.1:6379> hgetall con1:LegalIssueInformal
 1) "prakingViolation"
 2) "must bepaid before returning the car otherwise 100 dollar is charged"
 3) "contractRenewal"
 4) "automatically renewed inform us two days in advance at least"
 5) "deposit"
 6) "200 at check out time"
 7) "CollisionAndLiabilityInsurance"
 8) "no coverage if accident happens, creditcard will be charged"
 9) "discount"
10) "15% for AAA Members"
redis 127.0.0.1:6379> hgetall con1:LegalIssueFormal
 1) "prakingViolation"
 2) "IF (parkingValidation) AND −(DatePaid(ValidateFee)<(DateofReturn(car)) THEN (PayonReturn(RentFee+100))"
 3) "contractRenewal"
 4) "IF(InformDate(user)+2)<(DateofReturn(car))THEN (ConfirmRenewal())"
 5) "deposit"
 6) "IF (checkout)THEN (DepositPAid=200)"
 7) "CollisionAndLiabilityInsurance"
 8) "IF(accident) (THEN collisionCoverage=Null AND liability= Null AND charge(CreditCard))"
 9) "Discount"
10) "IF(AAAMember(user)) TEHN (PayonReturn((1-.015)*RentalFee))"
redis 127.0.0.1:6379> smembers con1:following
1) "s3"
2) "s2,"
3) "s1,"
redis 127.0.0.1:6379> smembers con1:followers
1) "context1"
redis 127.0.0.1:6379> hgetall con1:providerTrustDef
1) "rating"
2) "real"
3) "recommendaton"
4) "string"
redis 127.0.0.1:6379> hgetall con1:providerTrustVal
1) "rating"
2) "4"
3) "recommendaton"
4) "XXX Execllent"
redis 127.0.0.1:6379> hgetall con1:serviceTrustDef
 1) "safety"
 2) "string"
 3) "security"
 4) "string"
 5) "availability"
 6) "stringg"
 7) "reliability"
 8) "string"
redis 127.0.0.1:6379> hgetall con1:serviceTrustVal
 1) "safety"
 2) "finger print locking"
 3) "availability"
 4) "48 hours in advance"
 5) "reliability"
 6) "no break record"
```

Table 1.7

```
redis 127.0.0.1:6379> hgetall context1:info
1) "name"
2) "conx1"
3) "NodeType"
4) "context"
5) "following"
6) "con1"
7) "Rule"
8) "(Age>=18)"
redis 127.0.0.1:6379> hgetall context1:contextInfo
 1) "splocation"
 2) "enumerate"
 3) "SPAddress"
 4) "string"
 5) "SRLocation"
 6) "enumerate"
 7) "SRAddress"
 8) "string"
 9) "SRAge"
10) "int"
11) "SRLicenseExpiryDate"
12) "date"
redis 127.0.0.1:6379> hgetall context1:contextVal
 1) "splocation"
 2) "spl1"
 3) "SPAddress"
 4) "spadd1"
 5) "SRLocation"
 6) "srl1"
 7) "SRAddress"
 8) "sradd1"
 9) "SRAge"
10) "ageSr"
11) "SRLicenseExpiryDate"
12) "srdl1"
redis 127.0.0.1:6379> hgetall spl1
1) "spLocation"
2) "Montreal"
3) "sourceID"
4) "spc123"
5) "time"
6) "11:30"
7) "date"
8) "12/2/2013"
redis 127.0.0.1:6379> hgetall spadd1
1) "Address"
2) "15, 5th ave"
3) "sourceID"
4) "src123"
5) "time"
6) "11:35"
7) "date"
8) "12/2/2013"
redis 127.0.0.1:6379> hgetall srl1
1) "spLocation"
2) "Montreal"
3) "sourceID"
4) "src123"
5) "time"
6) "11:35"
7) "date"
8) "12/2/2013"
redis 127.0.0.1:6379> hgetall sradd1
1) "Address"
2) "25, 3rd ave "
3) "sourceID"
4) "src34"
5) "time"
6) "11:36"
7) "date"
8) "12/2/2013"
redis 127.0.0.1:6379> hgetall ageSr
1) "Age"
2) "31"
3) "sourceID"
4) "src12"
5) "time"
6) "11:45"
7) "date"
8) "12/2/2013"
redis 127.0.0.1:6379> hgetall srdl1
1) "SRDrivingLicenseExpire"
2) "12/12/2016"
3) "sourceID"
4) "src12"
5) "time"
6) "11:46"
7) "date"
8) "12/2/2013"
```

Figure 42: CASR Implementation in Redis (Snapshot(2))

6.2.7 Implementation Limitations

Redis is limited by its inability to provide complex structures. Consequently, we need to construct structures from the basic Redis structures. As mentioned before, structuring complex

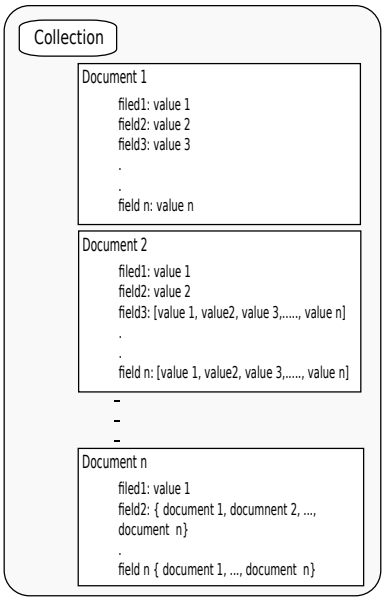


Figure 43: MongoDB Document-Oriented Store

structures requires several records of Redis to represent one element. For example, a function node in CASR registry is represented in eight records. These records are linked to each other by providing their key patterns. That is, all records start with the ID of the function. Yet, the operations to be performed on one complex structure record will involve too much work which might involve programming in either Java or Python with the client platform which can affect performance.

6.3 Implementation: MongoDB - Document-Oriented Store

In this section we discuss MongoDB features and their suitability for implementing the CASR generic model. MongoDB is an open source *document-oriented database*. It is easy to use and seems quite efficient in handling large volume of data as well.

6.3.1 MongoDB Features

Each record in MongoDB is called a *document*, which in turn is made up of a group of *fields* and their associated values. It can contain *embedded documents* with an overall size that does not exceed 16MB. The number of fields need not be the same in all documents. That is, each record can have a structure different from the structure of another record. Each document has a unique key by default and a secondary key can be assigned to some fields of the document. A *collection* is a pool of documents, which is equivalent to a table in SQL. The database supports all primitive types (Integer, String, Float), and arrays. Figure 43 illustrates MongoDB documents on one collection.

MongoDB is supported by ad-hoc query system that allows querying by a specific field of a document. MongoDB has high performance as it uses master/slave replication system. The slaves can only read and backup but all write operations is done by the master. The slaves are permitted to select another master in the case of master failure. In addition, MongoDB supports aggregation operation which is a benefit of using map reduce methodology [RWC12].

MongoDB users can display and use existing databases by using $show\ dbs$, and $use\ db$. If the command $use\ db$ is entered with a name of non-existing database, then MongoDB recognizes the user's desire to create a new one. Also, MongoDB provides several commands such as $db.Collection.insert()$ and $db.Collection.find()$ that help inserting and reading data. Also, MongoDB enhance readability of the retrieved data by the command

$$db.Collection.find().pretty()$$

which helps displaying data in an organized presentation. Figure 44 shows a sample MongoDB session that uses these basic operations..

One of the most important features of MongoDB is its indexing facility. Aside from the mandatory indexing that is automatically done by $MongoDBsystem\_id$, a secondary indexing facility is supported by MongoDB. This enables adding indexes on other fields of the documents.

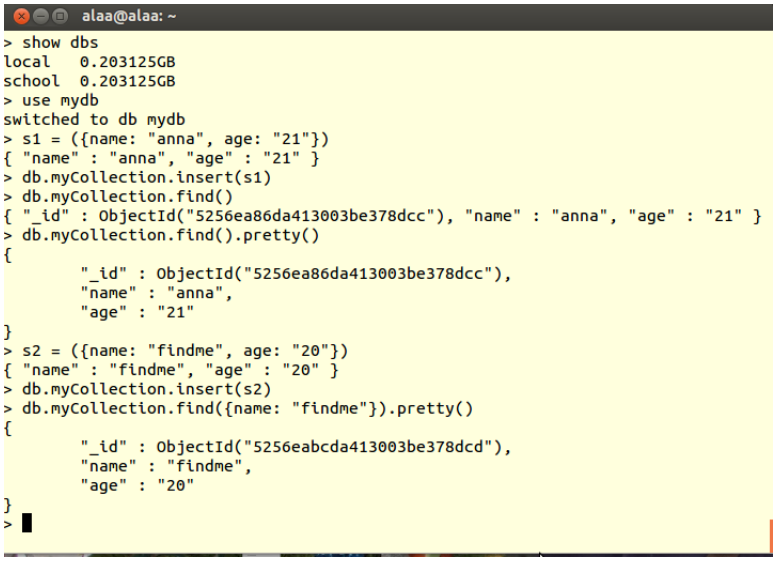


Figure 44: MongoDB Commands in Shell

The command $ensureIndex(fieldNames, features)$ specifies field names of one collection to be indexed and to clarify indexing features in feature parameters.

Example 16 *The* ConfiguredService *object in the generic model has* Followings *field to determine the parent of this node. If a query is required to bring all the* ConfiguredServices *of specific service provider* $SP1$*, then adding index on* Followings *field is required. This could be done in MongoDB by using the following command:*

$$db.cs.ensureIndex(Following : 1, unique : true, dropDups : true)$$

The first parameter of the function includes the field name which is Followings*, and the second parameter includes features. There features are* $dropDups$ *which is to drop any duplicates and* $unique$ *which is to create a unique index on* Followings*. This indexing method can be applied on several fields of the database. So, to get all the services provided by* $SP1$ *the following command is used:*

$$db.cs.find(Following : \text{“}SP1\text{”})$$

■

Although indexing can slightly affect the performance of MongoDB database, it comes with a reasonable profit. By introducing indexes we avoid the overhead that comes with querying large datasets. Therefore, we should choose to index the data set only if it is large and is most likely to be queried frequently [Cho13] [RWC12].

6.3.2 Design Consideration

MongoDB is a cross-platform document-oriented database system. It avoids the traditional table-based relational database structure in favour of documents with dynamic schemas. It's format makes the integration of data in certain types of applications easier and faster. Embedding and linking in MongoDB are two powerful features for designing a hierarchical structure. However, with these features there are two issues:

- Embedding and Linking: Embedding increases performance as one operation is required to get the data, but decreases query efficiency as MongoDB is not able to query field of embedded file. On the other hand, Linking decreases performance as more than one operation is performed for one query but query efficiency is enhanced by having no embedded fields queried.

- Document size: It should not exceed 16MB which is a lot of space. We pay attention to this restriction in embedding documents in MongoDB.

These features give the flexibility to a designer to think of different structures that can be best for a structure. For CASR registry, a naive approach would be to map all domains in the generic model to one collection, all sub-domains to another collection, and so on. Figure 45 shows this representation. Conceptually, this leads to a hierarchical model where each collection represents a level of the hierarchy. However, this design does not use any of MongoDB structuring best practices to build a good database. For example, embedding of data to enhance the efficiency of data retrieval and writing operations are not followed. Therefore, we

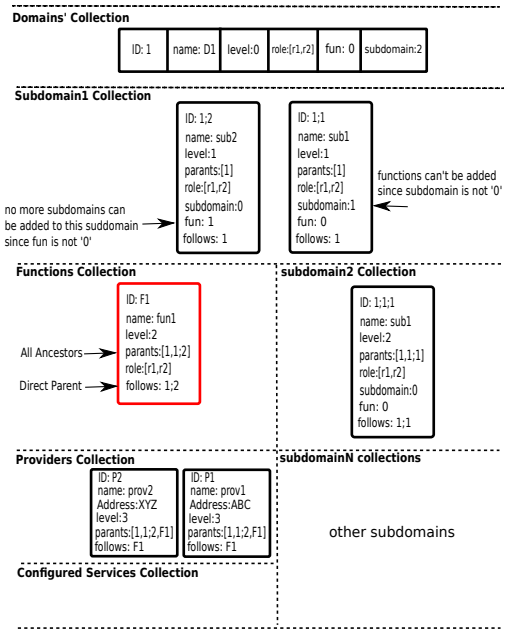


Figure 45: All Separated Collection Design

avoid this naive approach and look at two other methods.

Figure 46 shows another design called “All embedded”. This design, is the opposite of the previous one, which is to embed all records in one collection. That is, when mapping the generic model to this design all information from root(domain) to a leaf (*ConfiguredService*) will be stored in one document. That is, we embed all the sub-domains, functions, providers and configured services under one domain name in one document. Of course, one of the greatest flexibilities of document-based databases is the ability to build sub-documents. This facilitates the picturing of mapping the generic model to our structure as well as the understandability of the relationships among the documents. See Figure 46 in which two MongoDB documents are created from two branches of a domain node in the generic model. Besides being an intuitive structure, it is known for its high performance in retrieving the data. However, there are two great limitations to this approach. One limitation is the size of the MongoDB document might not allow this mapping. Although 16 MB is not small for a document, but for service data whose growth is unpredictable, it can cause problems. In addition, structuring

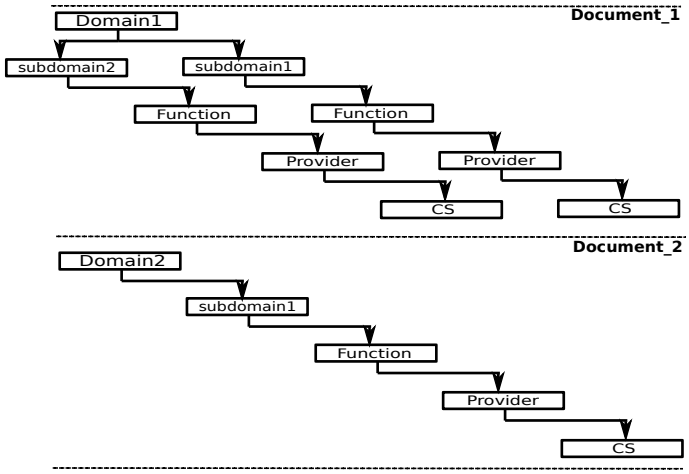


Figure 46: All Embedded

embedded documents is simple for two or three level sub-documents, but can become very complex and error-prone for ten level sub-documents. For these reasons, we conclude that mapping all information from domain down to configured service into a single document is to be avoided. That is, this model is not suitable for CASR.

Figure 47 shows the "three collection model". This model takes advantage of many MongoDB features and maps CASR generic model into three collections. First collection includes the domain information which includes sub-domain information represented as a sub-document of domain document. Same thing with the second collection that stores functions documents embedding providers documents. Finally, *ConfiguredServices* are mapped into a separate collection. *ConfiguredServices* should be in a separate collection as they represent most of the data and they are the most frequent queried nodes. That is, this collection not only will contain a large volume of information but also will be subjected to a high volume of querying. Considering the performance issue, it is better to keep *ConfiguredServices* as a separate collection, see Figure 47. This model is proper for implementation with some changes that are considered to enhance performance.

It seems that the best choice for mapping the generic model into MongoDB is to map the DKN part into one collection. In that collection, domain, sub-domain, and function parts are

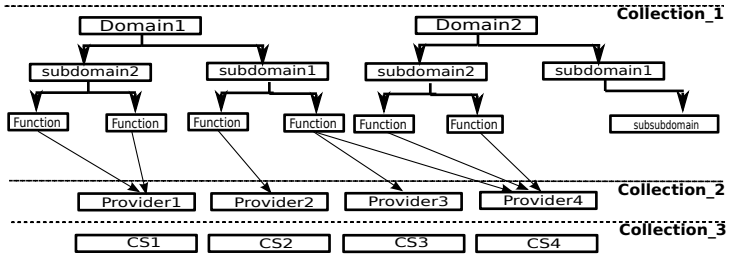


Figure 47: Three Collections Model

mapped into separate documents. That is, each node of the first levels of the generic model hierarchy represents a document in this collection. This structuring will enable fast browsing of the domain knowledge by the users without the need to look at more than one collection. Then, we represent the Provider entity of the generic model by representing service provider information in documents stored in the second collection. In this way, providers can be mapped to more than one function and *ConfiguredServices* in order to cut down on the repetition of data. *ConfiguredServices* are very rich to be structured in one document. Hence, the three parts service, contract and context of a *ConfiguredService* are separately mapped into three different documents. This increases the number of documents stored in *ConfiguredService* collection. Therefore, we decided to split *ConfiguredService* collection basing on the grandparent of *ConfiguredServices*. That is each function(grandparent) wraps all *ConfiguredServices* that provide this functionality including all documents related to their *ConfiguredServices*. That is, in the collection that stores the *ConfiguredService* we include their contract and context information, which are represented in separate documents of the collection. This collection structure is shown in Figure 48. In other words, function creates a collection under its ID. This collection includes *ConfiguredService* documents that is published as a grandchild of this function. Hence, there is no need to dive in to all services in order to get the ones that belong to a specific function. What is needed is to go through the collection that is named with the function ID. Hence, it is decided to manually cluster all *ConfiguredServices* based on functionsID. Each of these collections of the MongoDB design is explained below.

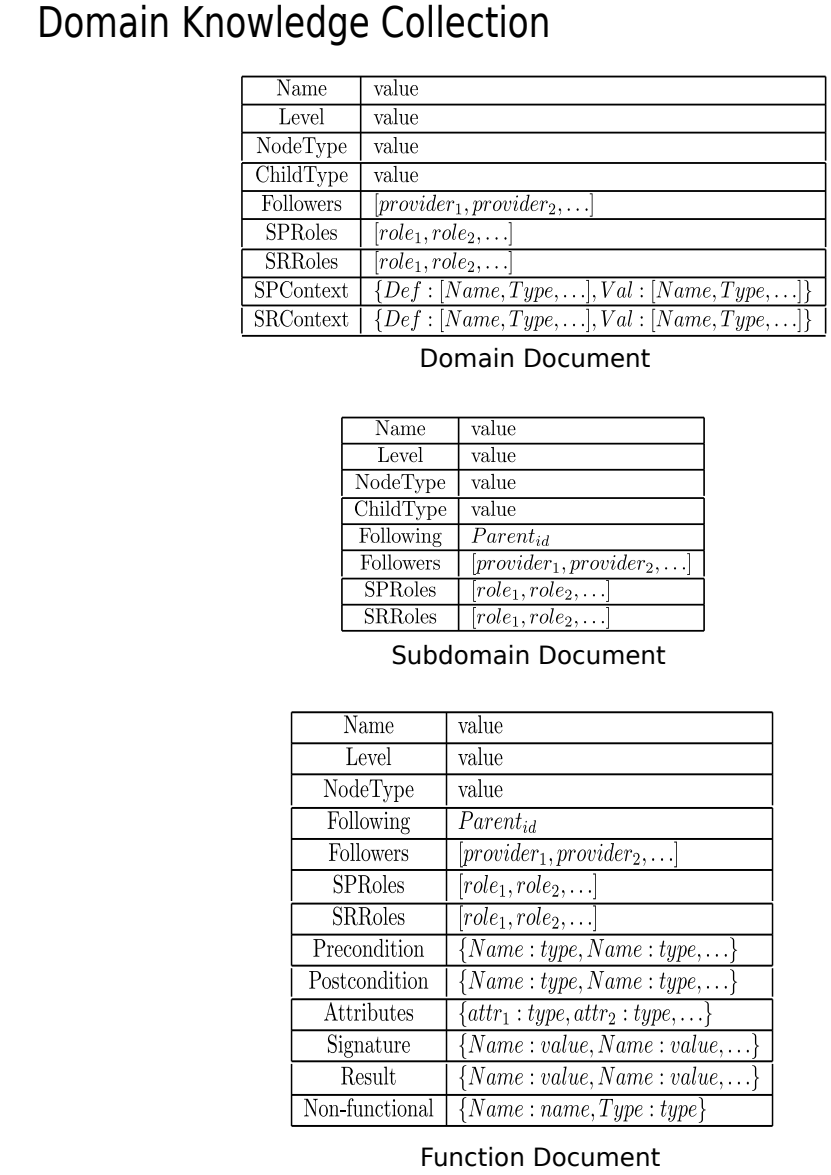


Figure 48: Domain Knowledge Implementation Model in MongoDB

6.3.3 MongoDB Implementation

1. **Domain Knowledge Collection**

   The DKN collection includes a single document for each of domain, sub-domain and function. One of the strong points of MongoDB is that it allows different documents within a collection to be structured differently. Exploiting this feature we structure domain document, sub-domain document, and function document differently. Figure 48 shows the

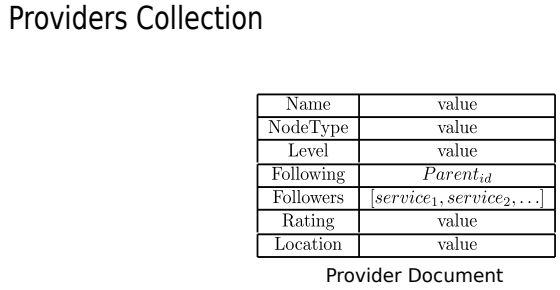


Figure 49: Provider Implementation Structure in MongoDB

mapping structure of the three first levels of generic model into MongoDB representation.

The domain document is structured in order to have a direct one-one mapping from the domain part in the generic model. Notice that both *SPContext* and *SRContext* fields are embedded documents. The advantage is that these sub-documents can be retrieved and used separately from the domain document. The rest of the fields in domain document are either records or atomic values. In sub-domain document and function document all fields are either records or atomic values. The mapping from the generic model preserves all consistencies between parents and children nodes. As an example, the set of roles in the SPRoles field of a sub-domain node will be a subset of the set of roles in SPRoles field of its parent.

2. **Provider Collection** The provider collection is a set of documents, where in each document we store one service provider in which its information defined in the generic model. The document structure for a service provider is shown in Figure 49. In the *Followers* field the identities of all provided services are wrapped. The *Followings* field refers to the parent node of the service, which defines what function this provider is using.

3. **Configured Service Collection** With MongoDB, it is decided to store *ConfiguredServices* in separate collections. Each collection represents the function that represents the service that the *ConfiguredServices* provides. Hence, there will exist many collections

for storing all *ConfiguredServices*. This requires the ability to construct a collection every time a function node is constructed. The collections is named by the function ID. A *ConfiguredService* in the generic model is represented in three documents stored in the collection that the *ConfiguredService* belongs to. These documents are used to represent service, contract, and context information defined in the generic model. Because many *ConfiguredServices* can have the same functionality, we assign the function ID to name *ConfiguredService*collection that stores all *ConfiguredServices* with this functionality. This design has the following advantages.

- Both contract and context parts of a *ConfiguredService* can be updated, either independently or jointly, without affecting the service part.
- When SR browses DKN and selects the desired function, the system can automatically find all *ConfiguredServices* that belong to this function by finding the collection distinguished by the function ID. The overall performance is enhanced by avoiding the concentrated load that comes when all the that are *ConfiguredServices* stored in one collection.

The three documents in the collection of *ConfiguredService* are linked through *Followings* and *Followers* fields. That is, each service document includes *Followers* and *Followings* information to know the ID of contract documents belonging to this service. Similarly, each contract document includes *Followers* field that includes the ID of context following this contract. It also includes *Followings* which defines the parent ID of this contract. All fields in service and contract documents are direct mappings of the respective *ConfiguredService* fields in the generic model.

The context document is richer than the other two documents, because we need many embedded documents in it. The *ContextInfo* part of a *ConfiguredService* contract is mapped to an embedded document that contains all dimensions as fields with their types

Fun_i Collection

Name	value
NodeType	value
Following	$Parent_{id}$
Followers	$[contract_1, contract_2, \ldots]$
Precondition	$\{Def : \{Name : type, Name : type, \ldots\}, Val : \{Name : value, Name : value, \ldots\}\}$
Postcondition	$\{Def : \{Name : type, Name : type, \ldots\}, Val : \{Name : value, Name : value, \ldots\}\}$
Attributes	$\{Def : \{attr_1 : type, attr_2 : type, \ldots\}, Val : \{attr_1 : value, attr_2 : value, \ldots\}\}$
Signature	$\{Def : \{Name : type, Name : type, \ldots\}, Val : \{Name : value, Name : value, \ldots\}\}$
Result	$\{Def : \{Name : type, Name : type, \ldots\}, Val : \{Name : value, Name : value, \ldots\}\}$
NonFunctional	$\{Name : value, Type : value, Discription : value\}$

Service Document

Name	value
NodeType	value
Following	$[Parent_{id_1}, Parent_{id_2}, \ldots]$
Followers	$[context_1, context_2, \ldots]$
ServiceTrust	$\{Def : \{Name : type, Name : type, \ldots\}, Val : \{Name : value, Name : value, \ldots\}\}$
ProviderTrust	$\{Def : \{Name : type, Name : type, \ldots\}, Val : \{Name : value, Name : value, \ldots\}\}$
LegalIssues	$\{FormalName : value, InformalName : value\}$

Contract Document

Name	value
NodeType	value
Following	$Parent_{id}$
ContextInfo	$\{id : InfoID, DIM_1 : type, DIM_2 : type, \ldots\}$
ContextRule	value
ContextValue	$\{id : valueID,$ $datetime : date/time,$ $clientID : CID,$ $providerID : PID,$ $serviceID : SID,$ $DIM_1 : [value, sourceID, datetime],$ $\vdots,$ $DIM_N : [value, sourceID, datetime]\}$

Context Document

Figure 50: Configured Service Implementation Model in MongoDB

as values. The *ContextRule* is a defined field in the contract that includes string value to represent the rule entered by SP. The *ContextValue* is mapped to an embedded document that contains fields and arrays. The fields *datetime, clientID, providerID,* and *serviceID*

within the embedded document of *ContextValue* have only atomic values. Each dimension of the context is modeled as an array structure, which wraps the information specific to each dimension in one memory block. Thus, all information regarding one dimension including *sourceID, date/time of collection,* and *value* of the dimension can be retrieved by the name of the dimension. The rationale for representing dimensions as arrays instead of embedded documents is to reduce the number of levels of document embedding. Increasing the number of levels of document embedding requires complex query processing and retrieval, and makes MongoDB's operations resource intensive. Thus, with the current structure, when an update operation is performed, only *lastupdate* field and the values of dimensions are updated with a single query.

6.3.4 Implementation Limitation

Using this implementation design can result in the following issues:

1. Some Collections will scale down. This is because of the decision we made for clustering the *ConfiguredServices* based on their common functionality. Some collections of *ConfiguredServices* might have only few records in it, because not too many SPs may publish such a service. Also when the SP of an existing *ConfiguredService* ceases to sustain providing a service it may be deleted from the system. Consequently, there is a possibility that a collection may end up being empty.

2. Collections Iterations: If there was a user request that requires some operation to be repeatedly applied to more than one *ConfiguredServices*, then we need to go through different collections to apply that operation. Consequently, substantial amount of work will have to be done to satisfy such user requests. It is also possible that maintenance operations might require going through different collections more frequently than others.

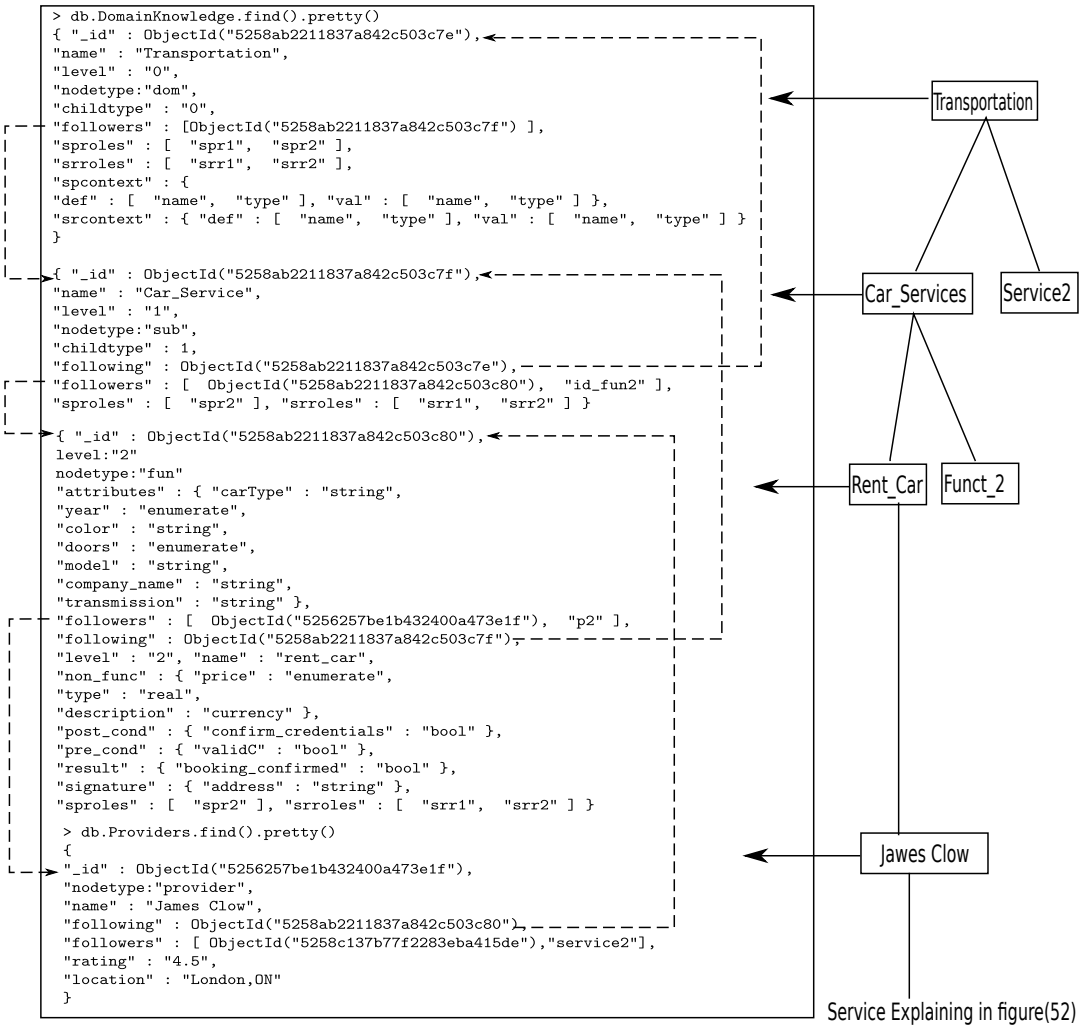


Figure 51: CASR Implementation in MongoDB (Snapshot(1))

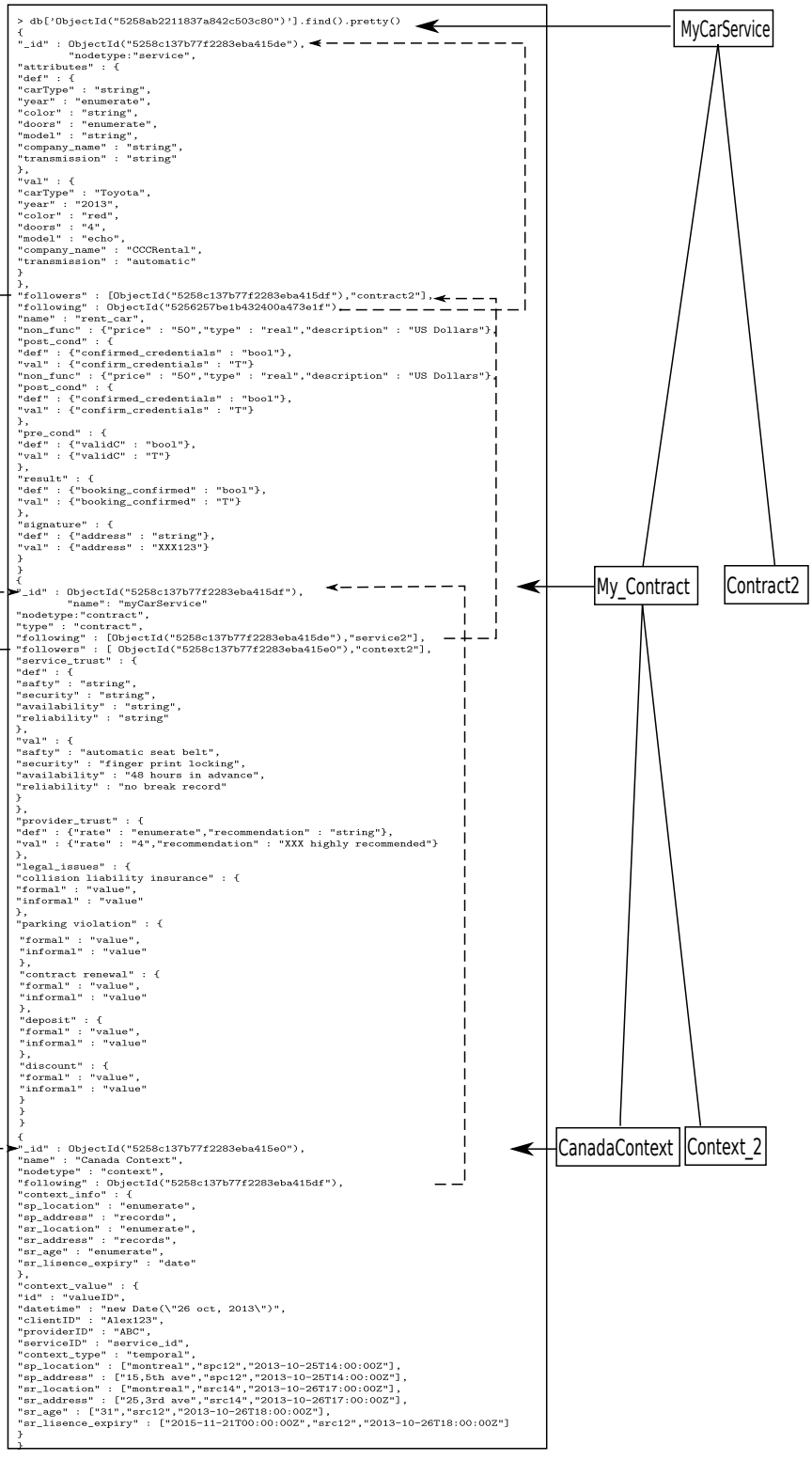


Figure 52: CASR Implementation in MongoDB (Snapshot(2))

6.4 Implementation: Hbase Column-Oriented Store

In this section we explain the features of Hbase and the possible mapping of CASR generic model into a Hbase. Hbase is an open source *column-oriented* database. It is designed by Google basing on *BigTable* techniques [Geo11], Hadoop and Distributed File System HDFS [Tay10].

6.4.1 Hbase Features

Hbase is designed to work with massive data by storing data in tables which are not similar to traditional tables of SQL databases. A table in Hbase is the big table that can expand vertically and horizontally. That is, Hbase welcomes to increase number of rows and columns as this is what it is designed for. The columns are like variables assigned for each row. Hbase supports the flexibility to provide different columns for each row. Rows are records of the table. Each table stores information based on key-value techniques. That is, a table of Hbase contains a bunch of keys (column qualifiers)-Values(cells) wrapped together under one name, which is called *Column Family* (CF). *Column Qualifier* (CQ) is a field within CF. *Row Key* is a unique key that differentiates a row from another. *Cell* stores an atomic value. To store this value or query it, three keys are needed. These keys are row key, column family, and column qualifier. The size of a cell could be from 10 to 50 MB [DKR13]. The bigger size a cell has, the better performance Hbase provides [Geo11]. *Version* of a cell is characterized by a time stamp. Whenever data is inserted or updated in a cell, the system stores a time stamp for this action. If time stamp is not specified when retrieving data, the system automatically returns the most recent data from the cell. Hbase does not provide indexing methods. By using only row key to query the data, the CFs that include data related to this row attached to their CFs and cells are provided. If row key is used with a specific CF in a query, then the CQs under this CF related this row are provided. The query can be narrowed to a cell if the row key, CF, and CQ are used in a query, because data is stored in the cell referenced by row key, CF and CQ. Figure 53 shows the

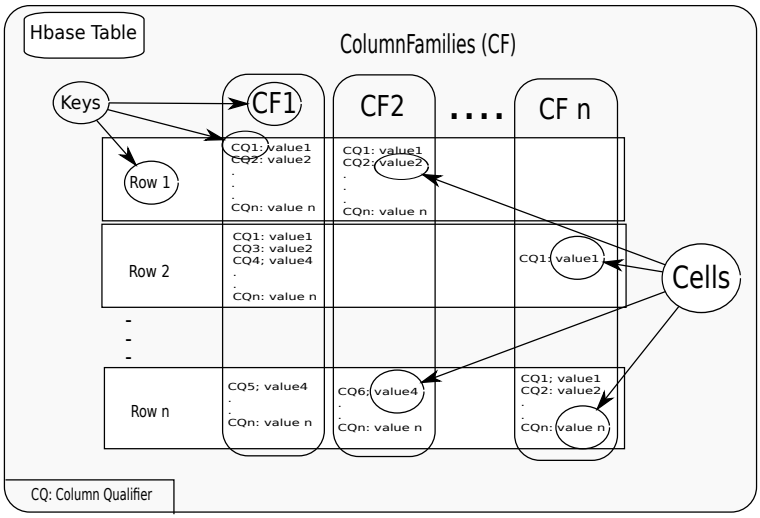


Figure 53: Hbase Column-Oriented Store

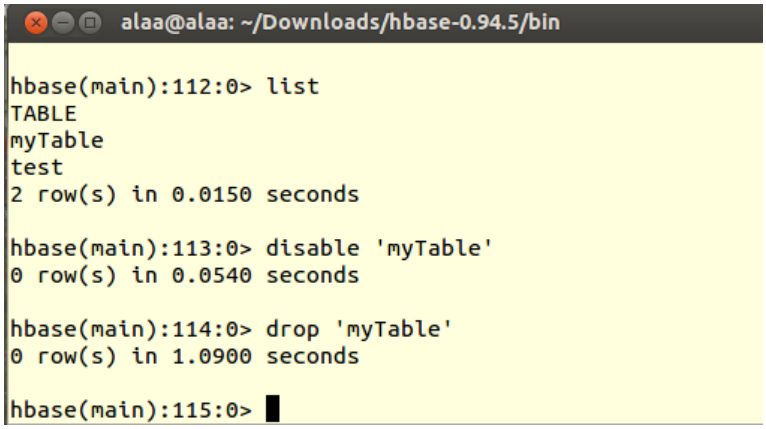


Figure 54: Disable and Drop Commands in Hbase Shell

column-oriented Hbase structure.

Hbase shell commands can serve with data definition such as alter, drop, and disable. Manipulating data is possible with operations such as count, delete, scan, get and put. To administer data clusters, the commands such as move, split, and disable_peer are used. To build our database we use the commands that are for defining and manipulating the data. Figure 54 shows a session using disabling and drop command. Figure 55 shows a session using Put command, and Figure 56 shows a session using Get command..

Hbase provides several features that meet the requirement of implementing CASR registry.

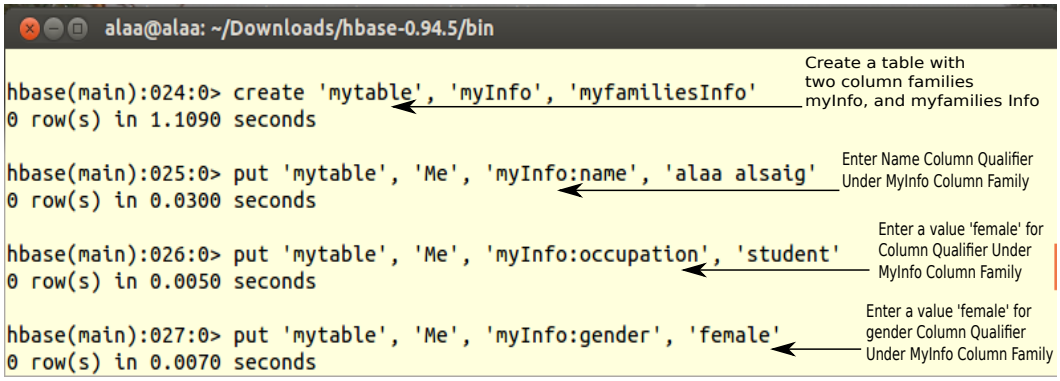


Figure 55: Put Commands by in Hbase Shell

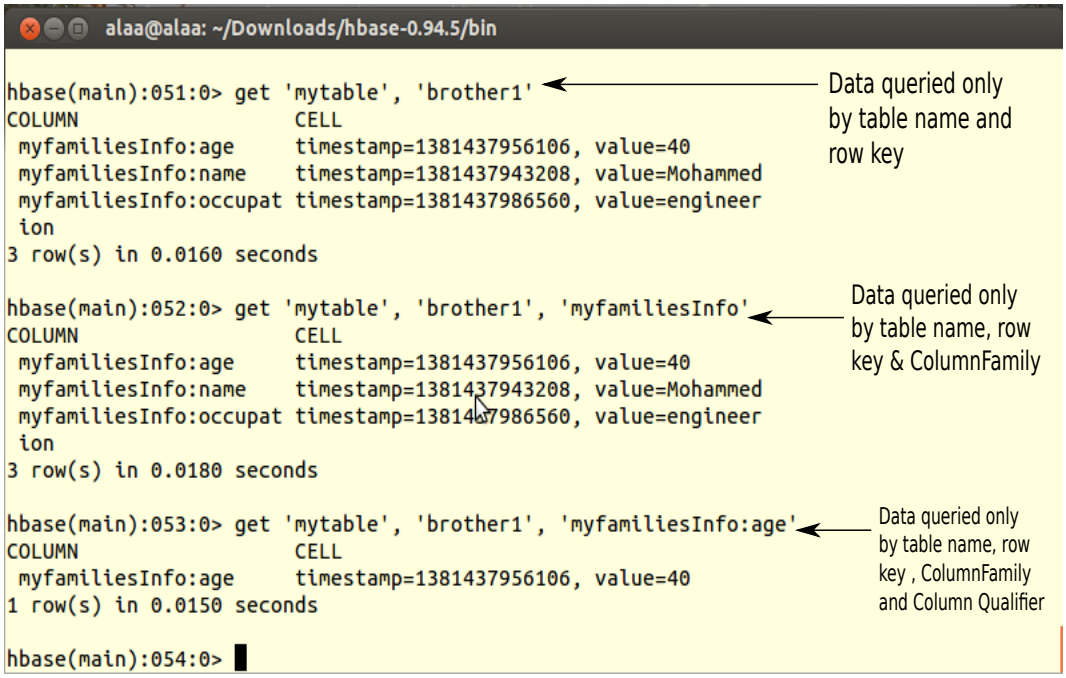


Figure 56: Get Commands in Hbase Shell

Supporting scalability, rich structure and querying all fields of the database are essential features to support flexibility and data expansion of CASR registry. Also, Hbase supports CRUD operations on several parts of the database which helps with modifying and updating database records [DKR13] [Geo11] [RWC12].

6.4.2 Design Consideration

When designing the database, it is important to be aware of the following restrictions:

- Limited Column Family: The number of column families should be small. It is better to keep a maximum of three CFs to optimize the performance. If it is necessary to have more than three CFs, it is better not to query more than three at any one time.

- Scale Down: Hbase does not perform well when it is not populated with massive data. Hbase can scale out but cannot scale down.

- Embedding: Hbase, similar to MongoDB, provides the flexibility to embed data and information inside each other. Embedding in Hbase is conceptual, but it is important for the designer to choose between "embedding all in one table" or "embedding in more than one table". This choice has a consequence in how data is stored. This is because in Hbase information of one table is stored in one region. As an example, we show two different embeddings for the same database in Figure 57 and Figure 58.

- Column Qualifier Benefits: There is no limitation on the number of column qualifiers. It is better to use column fields as stored information because this increases efficiency. That is, CQ names can themselves provide information to a row in the table. For example, for a student taking courses, instead of having the field $RegisteredCourses$ for each student that will include all IDs of courses taken by a specific student, it is better to construct a table that has CF called courses and under this CF each column represents a course name taken by a specific student. Each student does not have to have the same CQ name. Hence, courses taken by a student can easily be retrieved by just providing the student ID, which is the row key, and CF name(courses). Then all CQ(courses names) belong to this student under this CF is provided.

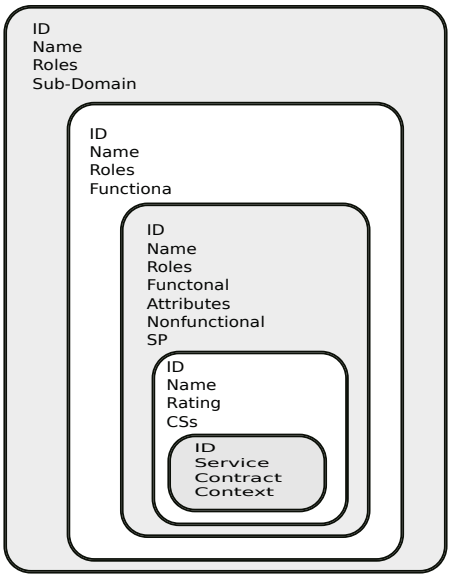


Figure 57: Embedded Columns In Hbase

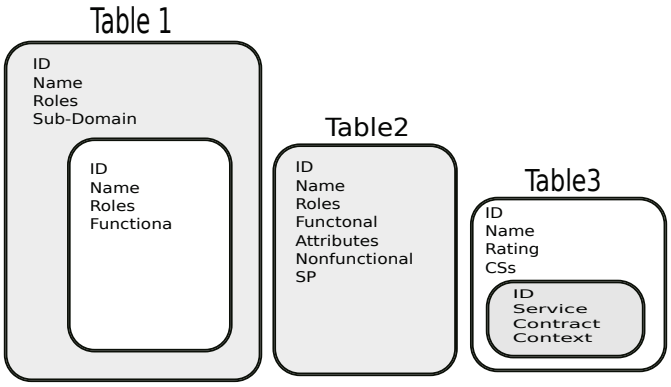


Figure 58: Embedded Columns In Hbase

To map the generic structure to Hbase, it is difficult to structure based on the classification DKN, Provider, *ConfiguredService*, and context. The nature and features of Hbase has driven us to classify the generic model basing on common and similarity of nodes structure. Figure 59 shows the nodes mapped to the tables that are constructed by Hbase. The following sections explain each table.

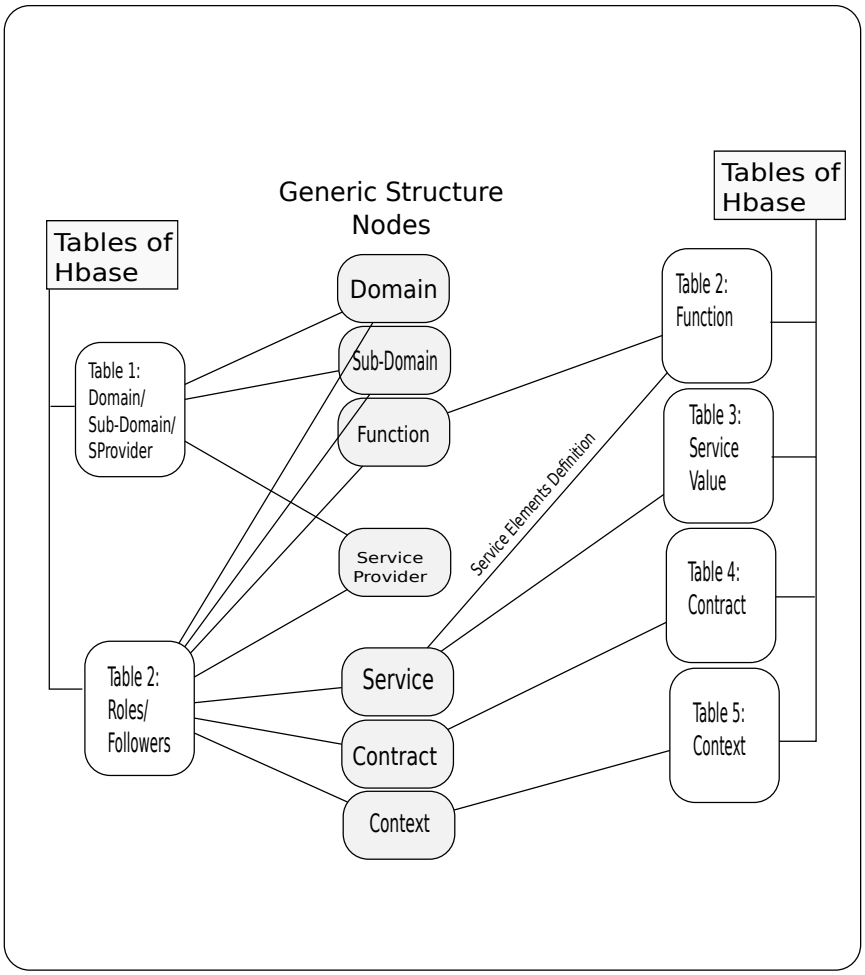


Figure 59: Mapping The Generic Model Nodes to Hbase

6.4.3 Table1: Domain, Sub-Domain and Providers in Hbase

We structure the domain, sub-domains and service providers in Figure 60. This table includes three CFs which are Info, SPC/SRC DEF, and SPC/SRC Value. The Info CF wraps all the columns that provide information to the row key that can be a key for a domain, sub-domain or a SP. This information is represented in column qualifiers which are $Name$, *Level*, *NodeType*, *ChildType*, and *Followings*. All these columns are defined to represent the fields defined in the generic model. As mentioned in previous chapters, the $Name$ field is the name of the node, the *Level* field helps to define in which level of the hierarchy this node is, *NodeType* defines the type of the node, *ChildType* is to strict the children nodes of this node to the defined type,

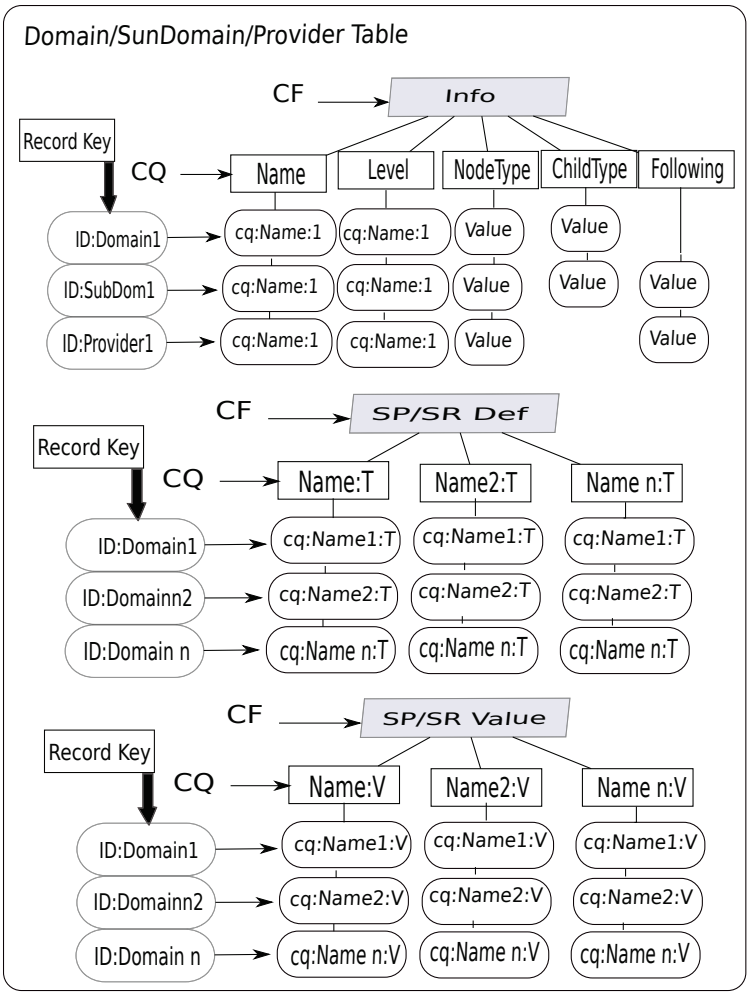
and *Followings* is to recognize the parent of this node. The Level information gives the ability to display the DKN and other information in an organized manner,

Figure 60 highlights that the records of the table do not have to be of the same length. Besides, any cell that is not assigned to NULL value will not exist in the table. The SPC/SRCDef CF is the family that includes every domain SPContext and SRContext dimensions and their types. The last column family is the SPC/SRCVal includes the same dimensions defined in the SPC/SRCDef associated with their values adding to it fields time, data and sourceID. The SPC/SRC context are filled with values until SR instantiates a session with one of the *ConfiguredServices*. Hence, the SP information is gathered to populate SPC/SRC context values. The system updates the values whenever another sessions starts. As versioning is accumulated by Hbase, we decided to set it to zero copy of versions with this column family. There is no need to fill the memory with redundant data.

The reason for putting all these three nodes together is that the domain, sub-domain, provider nodes have similar and considerably simpler structures from the other nodes in the generic model. The *ConfiguredService* and function nodes include several elements, with each element in it having many fields and their values. Hence, it is decided to change the organization of structuring, yet we preserve the concept of providing DKN to SR. This is achieved by including all the fields in a table that keeps the structure of generic model. This table is explained in Section 6.4.8.

6.4.4 Table2: Function in Hbase

We map the function node defined in the generic model to Hbase by constructing one table called Function. The Function Table is to map the function node of the generic model into Hbase. The table includes Info CF. Similar to table1, the CF Info includes the information defined in the generic model including *Name*, *Level*, *NodeType*, and *Followings*. Also, the info CF has additional information related to attribute, non-functional, preCond and postCond CQs.



Hbase.pdf

Figure 60: Table 1, Domain, Sub-Domain and providers Table in Hbase

These CQs are associated with the row key of these records that are stored in the same table. That is, according to the generic model of function structure, function includes four elements which are Functional, Non-functional, Signature and Result. Each one of these elements is constructed with a set of fields which represent Pre-Condition, Post-Condition or attributes fields. Each field is associated with the data type of this field. As each element includes several fields, each element needs a row to be represented or may be put in a separate table. However, in Hbase we can create many different CQs. So, we decided to include five CQs added to their Info CQs. These CQs are Functional, Non-functional, Attributes, Signature and Result. The cell of each of these includes the row key of this element. Thus, there will be a row key for

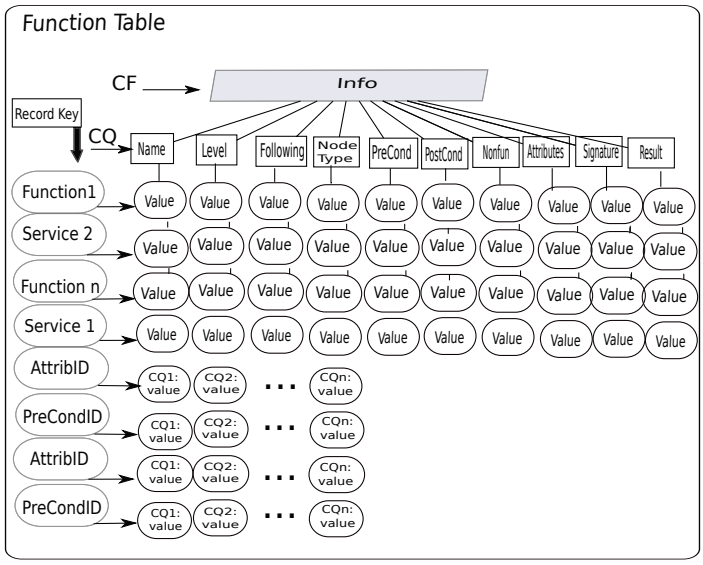


Figure 61: Table2: Function Table in Hbase

functional element, a row key for attribute, a row key for attributes element, a row key for signature and a row key for result element. Every time function is queried Hbase gets these IDs from function row and dives into the same table to get the rest of the information using the discovered keys. This does not affect the performance of Hbase as we are only scaling up. This approach is preferable than creating more CFs. figure 61 shows the mapping of function from the generic model to Hbase table.

The service node in Figure 59 is mapped to the function table Table.2. which is because of the similarity between service definition and function structure. The service definition is exactly structured with the same elements of the function. Moreover, function includes other attributes from other services that are their grandchildren. This is explained with examples in Section 6.2. As Hbase provides the flexible schema for records of the same table and welcomes scaling up the data, we are motivated to include all services definition in one table. This helps to have only a fewer number of tables and hence a fewer number of CFs, which in turn provides better performance. Also, as the load on service table is heavy, in terms of querying, it is decided to lessen the work on service table (Table3) by including their definition in function

table(Table2).

6.4.5 Table3: Services in Hbase

We map the service node constructed as part of the *ConfiguredService* in the generic model in Hbase by constructing one table for service. This table is similar to the table of function. Yet it includes only service records and the values of their elements without including their type. The table includes the CF Info that includes the $Name$, *NodeType*, and following fields that are defined earlier. The IDs of PreCon, PostCond, NonFunctional, Attributes, Signature, and result are added under this CF as well. Info CF also includes a field called *grandParent* which is equal to the provider's following value. This is to help the system find all *ConfiguredServices* provided under a specific function without going through SP level. That is, if we want to check all the services belonging to one function, we have to query the field *grandParent* to let the system to recognize all grandchildren of a specific function. This eases the work on CASR registry to query the services belonging to a specific function by skipping the level of provider.

The rows that represent pre-conditions and post conditions represent a list of pre-conditions and post-conditions of a specific values associated with the value of this condition. Non-Functional is mapped to three CQs which are the names of the parameter, its value and the description field that describes the parameter. Attributes, Signature and Result include list of fields associated with their values.

6.4.6 Table4: Contract in Hbase

We map the contract component of the generic model into one table. The contract includes Service trust definition and value, Provider trust definition and value and Legal issues. The service trust needs to have a field that includes two values which are the value and its field. The Legal issue part of the contract needs a field that is associated with two values, which are the formal and the informal representation of the rule. Therefore, we decided to use

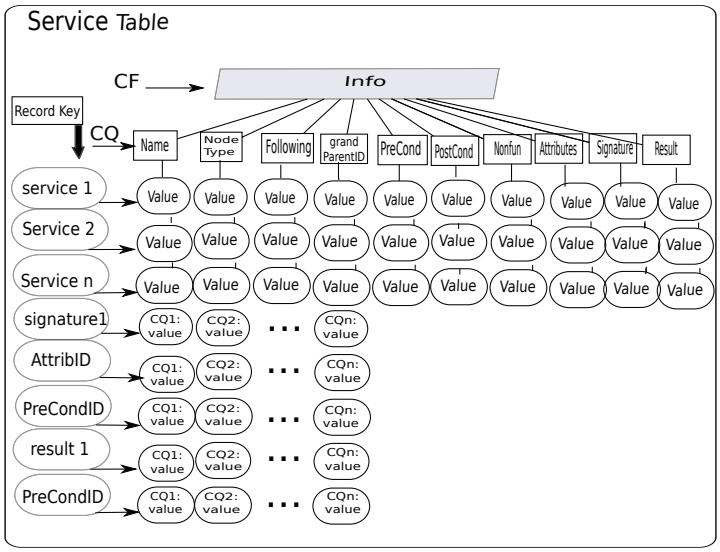


Figure 62: Table 3: Service Table in Hbase

the same methods that was used with service. The contract table includes one CF called Info. This Info includes the *Name*, *NodeType*, *Followings*ID, serviceTrustID, providerTrustID, Rule1ID, Rule2ID,...,RulenID. These IDs represent row keys of those elements. To retrieve the information of any of those elements the contract ID is enough to have them reachable. All records are stored in the same table Figure 63. The *Followings* is separated in a single records because it represents all IDs of services the contract belongs to. This represent the generic model definition for *Followings* field in *Contract* node.

6.4.7 Table5: Context in Hbase

We designed the Service Context Model using the CF feature in Hbase. This is shown in Figure 64 in which we have named the column family as Context Family. The Context Family is mapped to a set of dimension names defined as CQ. Also, these CQs include other columns that provide information to the *ContextValue*. Actually, columns include the data that are related to one or more rows and rows include data that are related to the dimensions. As illustrated by Figure 64, only the *ContextValue* needs all the fields represented by columns. This results in rows with different lengths. The third row key is *ContextRule* which does not need any data

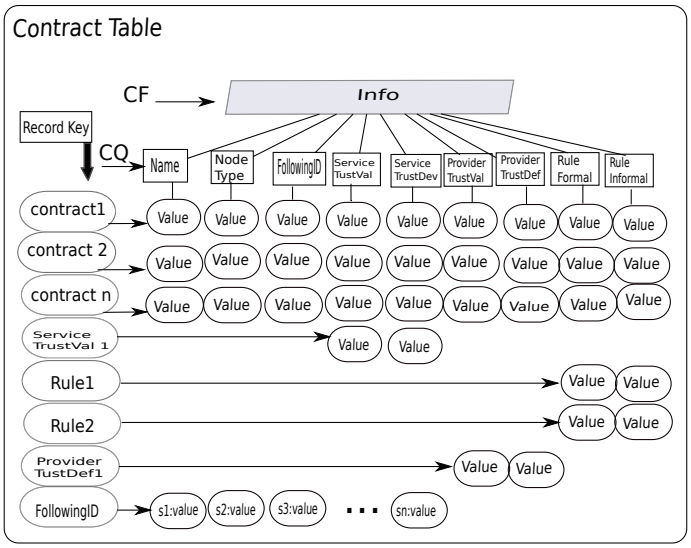


Figure 63: Table4: Contract Table in Hbase

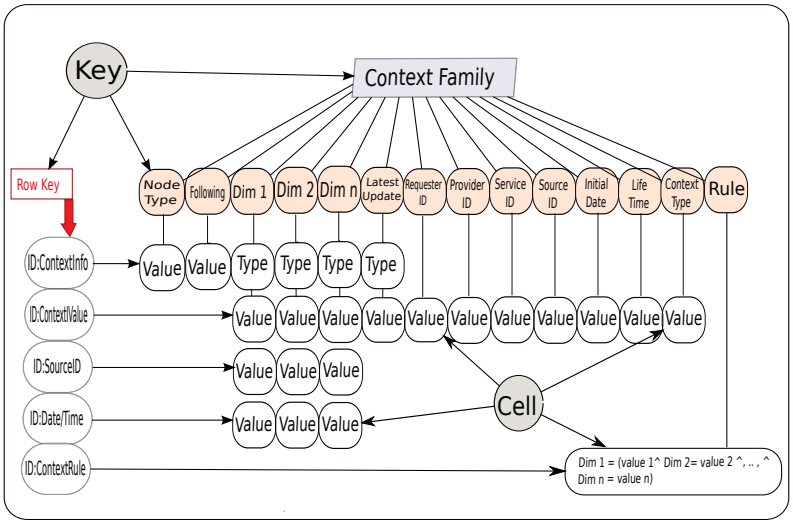


Figure 64: Table5: Context Table in Hbase

of column defined in the structure. To map the *ContextRule* element of context defined in the generic model into Hbase, another CQ is defined. This CQ is called Rule. *ContextRule* will not reserve extra space by defining one CQ in its record. Actually, we have the ability in Hbase to include any number of CQs which gives the flexibility to have one table encompasses several elements with different data structures. These features are the most useful during mapping. Not only context information but also all other nodes of the generic model use this feature.

6.4.8 Table6: Followers/SRSPRole in Hbase

The relationships among records that are stored in the defined Hbase tables are represented in another Hbase table. In the generic model, every node includes SPRole and SRRole. These two lists are included with domain, sub-domain and function nodes of the generic model to control access on services. Also, *Followers* fields is defined in the generic model to include children ID of each node. Driven by Hbase structure, we decided to separately map these information in a new table. This decision is motivated by the need to ease the work on building up the skeleton of the DKN. The relation *Followers* is represented as in Table5 of Figure 64. This table includes three CFs, which are Followers, SPRoles and SRRoles. For the column families SRRoles and SPRoles, every domain, sub-domain and function have CQs with each CF. The CQs represent the roles of SP under the family SPRoles, and represent roles of SR when they belong to SRRoles family. However, the column family Followers encompasses information on every domain, sub-domain, function, provider, service, contract and context in the CASR registry. The CQs of each record in Followers represent IDs of other records that follow them. In contract representation the *Followers* table includes any record ID related to its context. That is, the context table includes several records that represent one context information. Therefore, all these IDS included in the *Followers* list of the contract. Hence, the contract is linked to all the details of its context.

6.4.9 Limitations

Hbase is designed to withstand massive data with the ability to scale up. However, the performance is limited by the number of column families. It is not preferred to query more than two or three CFs at the same time as this increases the load of Hbase and hence loses performance. The community of Hbase developers have suggested to keep the number of CF low in the database. Following this suggestion we have devised our mappings from the generic model

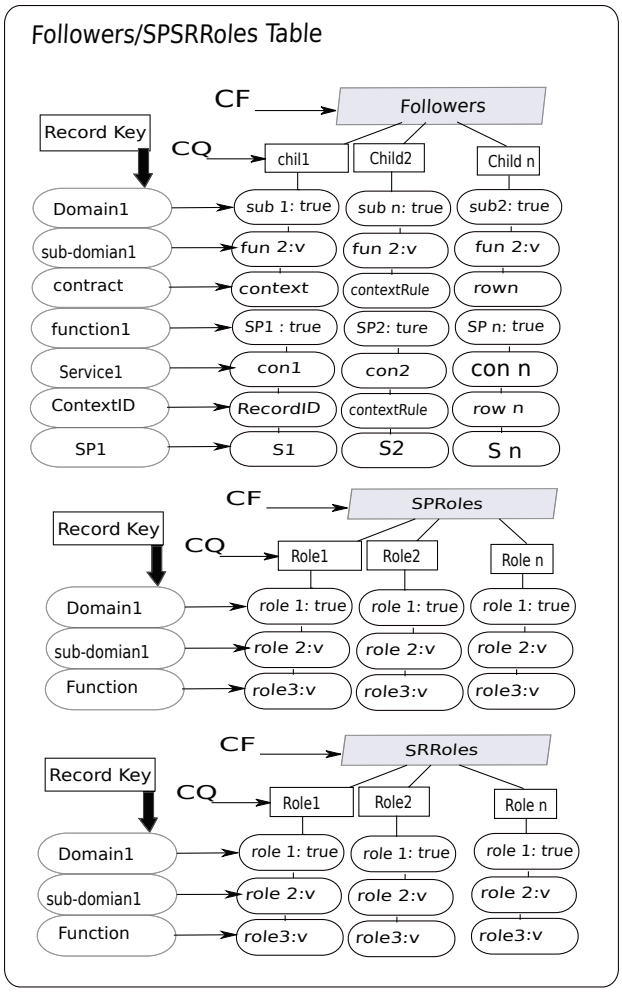


Figure 65: Table 6: Followers/SPSR Roles in Hbase

to Hbase tables. However, the structure of CASR registry is rich and each component needs the ability to store fields and values. Hence to map the generic model into Hbase we needed seven CFs among all tables.

```
hbase(main):076:0> get 'table1', 'dom1'
COLUMN                CELL
 SPSRDEF:ID           timestamp=1381761977002, value=SPDEF
 SPSRDEF:SRID         timestamp=1381761994780, value=SRDEF
 info:Name            timestamp=1381761894814, value=Transportation
 info:NodeType        timestamp=1381761923624, value=dom
 info:childType       timestamp=1381761933051, value=0
 info:level           timestamp=1381761905823, value=0
6 row(s) in 0.0200 seconds

hbase(main):077:0> get 'table1', 'SPDEF'
COLUMN                CELL
 SPSRDEF:Name         timestamp=1381762563645, value=string
 SPSRDEF:email        timestamp=1381762575662, value=string
 SPSRDEF:location     timestamp=1381762570296, value=string
 SPSRDEF:rating       timestamp=1381762582801, value=string
 SRSPVal:Name         timestamp=1381762480219, value=alaa
 SRSPVal:email        timestamp=1381762498594, value=alaa@hotmail.com
 SRSPVal:location     timestamp=1381762492035, value=Montreal
 SRSPVal:name         timestamp=1381762128951, value=string
 SRSPVal:rating       timestamp=1381762505356, value=5/5
9 row(s) in 0.0170 seconds

hbase(main):078:0> get 'table1', 'SRDEF'
COLUMN                CELL
 SPSRDEF:Name         timestamp=1381762663307, value=string
 SPSRDEF:email        timestamp=1381762670421, value=string
 SPSRDEF:location     timestamp=1381762655370, value=string
 SRSPVal:Name         timestamp=1381763132867, value=abc
 SRSPVal:email        timestamp=1381763176408, value=abc@whatever.com
 SRSPVal:location     timestamp=1381763153968, value=xxxx

8 row(s) in 0.0230 seconds

hbase(main):010:0> get 'table1', 'sub1'
COLUMN                CELL
 info:Following       timestamp=1381763370354, value=dom1
 info:NodeType        timestamp=1381763397572, value=sub
 info:childType       timestamp=1381763406700, value=1
 info:level           timestamp=1381763378414, value=1
 info:name            timestamp=1381763352913, value=carServices
5 row(s) in 0.0230 seconds

hbase(main):011:0> get 'table1', 'sp1'
COLUMN                CELL
 info:Following       timestamp=1381763480824, value=fun1
 info:NodeType        timestamp=1381763465126, value=sp1
 info:name            timestamp=1381763443722, value=alaa
3 row(s) in 0.0100 seconds

hbase(main):010:0> get 'table1', 'sub1'
COLUMN                CELL
 info:Following       timestamp=1381763370354, value=dom1
 info:NodeType        timestamp=1381763397572, value=sub
 info:childType       timestamp=1381763406700, value=1
 info:level           timestamp=1381763378414, value=1
 info:name            timestamp=1381763352913, value=carServices
5 row(s) in 0.0230 seconds
```

Table1

```
hbase(main):001:0> get 'table2', 'fun1'
COLUMN                CELL
 info:AttrinutesID    timestamp=1381763684288, value=AttID
 info:Name            timestamp=1381763604136, value=Rent_Car
 info:NodeType        timestamp=1381763631233, value=fun
 info:NonfunID        timestamp=1381763720041, value=nfID
 info:PostConID       timestamp=1381763661414, value=PostID
 info:PreConID        timestamp=1381763651847, value=PreID
 info:ResultID        timestamp=1381763708802, value=resID
 info:SignatureID     timestamp=1381763697736, value=sigID
 info:level           timestamp=1381763619086, value=2
9 row(s) in 0.6680 seconds

hbase(main):017:0> get 'table2', 'AttID'
COLUMN                CELL
 info:CarType         timestamp=1381769828069, value=string
 info:Color           timestamp=1381769844576, value=string
 info:CompanyName     timestamp=1381769903259, value=string
 info:Doors           timestamp=1381769869004, value=int
 info:Model           timestamp=1381769963113, value=string
 info:Tranmission     timestamp=1381769888479, value=string
 info:Year            timestamp=1381769861333, value=int
7 row(s) in 0.0400 seconds

hbase(main):019:0> get 'table2', 'nfID'
COLUMN                CELL
 info:description     timestamp=1381763825783, value=string
 info:price           timestamp=1381763814830, value=real
2 row(s) in 0.0120 seconds


hbase(main):004:0> get 'table2', 'preID'
COLUMN                CELL
 info:ValiDC          timestamp=1381770386953, value=bool
 info:ValiDD          timestamp=1381770393459, value=bool
2 row(s) in 0.0260 seconds

hbase(main):005:0> get 'table2', 'postID'
COLUMN                CELL
 info:ClientConfirm   timestamp=1381770408068, value=bool
1 row(s) in 0.0060 seconds


hbase(main):016:0> get 'table2', 'sigID'
COLUMN                CELL
 info:address         timestamp=1381763839898, value=string
1 row(s) in 0.0090 seconds

hbase(main):016:0> get 'table2', 'sigID'
COLUMN                CELL
 info:bookingConfirm         timestamp=1381763839898, value=bool
1 row(s) in 0.0090 seconds
```

Table2

```
hbase(main):001:0> get 'table3', 's1'
COLUMN                CELL
 info:AttributesID    timestamp=1381764264941, value=AttID
 info:Following       timestamp=1381764222673, value=sp1
 info:NodeType        timestamp=1381764199141, value=service
 info:NonfuncID       timestamp=1381764361715, value=nfID
 info:PostCondID      timestamp=1381764283162, value=postID
 info:PreCondID       timestamp=1381764276048, value=preID
 info:ResultID        timestamp=1381764318388, value=resID
 info:SignatureID     timestamp=1381764305445, value=sigID
 info:name            timestamp=1381764176176, value=myservice
10 row(s) in 0.6580 seconds

hbase(main):003:0> get 'table3', 'preID'
COLUMN                CELL
 info:ValidC          timestamp=1381764443117, value=T
 info:ValidD          timestamp=1381764439792, value=T
2 row(s) in 0.0120 seconds

hbase(main):008:0> get 'table3', 'postID'
COLUMN                CELL
 info:ClientConfirm   timestamp=1381764461910, value=T
1 row(s) in 0.0050 seconds


hbase(main):018:0> get 'table3', 'AttID'
COLUMN                CELL
 info:CarType         timestamp=1381770013238, value=toyota
 info:Color           timestamp=1381769985771, value=Red
 info:CompanyName     timestamp=1381769920277, value=CCCrenatl
 info:Model           timestamp=1381769968646, value=Echo
 info:Tranmission     timestamp=1381769932348, value=Automatic
5 row(s) in 0.0250 seconds

hbase(main):001:0> get 'table3', 'nfID'
COLUMN                CELL
 info:description     timestamp=1381764408242, value=US Dollar
 info:price           timestamp=1381764386922, value=50
2 row(s) in 0.6310 seconds

hbase(main):006:0> get 'table3', 'sigID'
COLUMN                CELL
 info:address   timestamp=1381764482864, value=Montreal, QC
2 row(s) in 0.0300 seconds

hbase(main):007:0> get 'table3', 'resID'
COLUMN                CELL
 info:bookingconfirem timestamp=1381764583910, value=T
1 row(s) in 0.0080 seconds
```

Table3

Figure 66: CASR Implementation in Hbase (Snapshot(1))

Table4

```
hbase(main):019:0> get 'table4', 'con1'
COLUMN              CELL
 info:FollowingID    timestamp=1381768396003, value=FollowingID
 info:NodeType       timestamp=1381768348648, value=contract
 info:formalID       timestamp=1381768422757, value=formID
 info:informalID     timestamp=1381768413318, value=infID
 info:trustDevID     timestamp=1381768465345, value=trddefID
 info:trustValID     timestamp=1381768457884, value=trdvalID
7 row(s) in 0.0370 seconds

hbase(main):018:0> get 'table4', 'trddef','info'
COLUMN              CELL
 info:reliability    timestamp=1381768647044, value=string
 info:safety         timestamp=1381768616727, value=string
 info:security       timestamp=1381768625598, value=string
 info:vailability    timestamp=1381768634337, value=string
4 row(s) in 0.0190 seconds

hbase(main):030:0> get 'table4', 'infID'
COLUMN              CELL
 info:deposit        timestamp=1381771271656, value=200 at a time of checkout
 info:discount       timestamp=1381771416427, value=15% for AAA member
2 row(s) in 0.0070 seconds

hbase(main):029:0> get 'table4', 'formID'
COLUMN              CELL
 info:deposit        timestamp=1381771496216, value=IF (Checkout) THEN (Deposit
                     Paid=200)
 info:discount       timestamp=1381771638234, value=IF (AAAMember(user)) THEN (
                     PayOnReturn((1-0.15)*RentalFee)
2 row(s) in 0.0080 seconds
```

Table6

```
hbase(main):029:0> get 'table6', 'dom1'
COLUMN              CELL
Followers:sub1      timestamp=1381211496216, value= 1.1
Followers:sub2      timestamp=1381761496216, value= 1.2
SPRoles:r1      timestamp=1381771496216, value= 1.3
SPRoles:r2      timestamp=1381231496216, value= 1.4
SPRoles:r3      timestamp=1381701496216, value= 1.5
SRRoles:r11      timestamp=1381771496216, value= 1.6
SRRoles:r12      timestamp=1381771496216, value= 1.7
SRRoles:r13      timestamp=1381771496216, value= 1.8

hbase(main):029:0> get 'table6', 'sub1'
COLUMN              CELL
Followers:fun1      timestamp=1381771496216, value= 1.1
Followers:fun2      timestamp=1381771496216, value= 1.2
SPRoles:r1      timestamp=1381771496216, value= 1.3
SPRoles:r2      timestamp=1381771496216, value= 1.4
SRRoles:r11      timestamp=1381771496216, value= 1.6
SRRoles:r12      timestamp=1381771496216, value= 1.7


hbase(main):029:0> get 'table6', 'fun1'
COLUMN              CELL
Followers:sp1      timestamp=1381771496216, value= 1.1
Followers:sp2      timestamp=1381771496216, value= 1.2
SPRoles:r1      timestamp=1381771496216, value= 1.3
SPRoles:r2      timestamp=1381771496216, value= 1.4
SRRoles:r13      timestamp=1381771496216, value= 1.8
```

Table5

```
hbase(main):029:0> get 'table5', 'conx1:ContextInfo'
COLUMN              CELL
contextFmaily:name     timestamp=1381771496216, value= context1
contextFmaily:NodeType     timestamp=1381761496216, value= context
contextFmaily:splocation     timestamp=1381771496216, value= string
contextFmaily:srlocation     timestamp=1381781496216, value= string
contextFmaily:spAddress     timestamp=1381771496216, value= string
contextFmaily:srAddress     timestamp=1381781496216, value= string
contextFmaily:srage     timestamp=1381771496216, value= enumerate

hbase(main):029:0> get 'table5', 'conx1:ContextValue'
COLUMN              CELL
contextFmaily:splocation     timestamp=1381771496216, value= Montreal, QC
contextFmaily:srlocation     timestamp=1381781496216, value= Lasalle, QC
contextFmaily:spAddress     timestamp=1381771496216, value= 15, 5th Ave
contextFmaily:srAddress     timestamp=1381781496216, value= 25, 3rd Ave
contextFmaily:srage        timestamp=1381771496216, value= 31
contextFamily:SPID     timestamp=1381781496216, value= sp1
contextFamily:ServiceID     timestamp=1381781496216, value= s1
contextFamily:latestUpdate  timestamp=1381781496216, value= 11:20
contextFamily:date     timestamp=1381781496216, value= 12/12/13

hbase(main):029:0> get 'table5', 'conx1:splocation'
COLUMN              CELL
contextFmaily:time      timestamp=1381771496216, value= 11:30
contextFmaily:date      timestamp=1381771496216, value= 12/12/13
contextFmaily:sourceID      timestamp=1381771496216, value= spc12

hbase(main):029:0> get 'table5', 'conx1:srlocation'
COLUMN              CELL
contextFmaily:time      timestamp=1381771496216, value= 11:33
contextFmaily:date      timestamp=1381771496216, value= 12/12/13
contextFmaily:sourceID      timestamp=1381771496216, value= src13

hbase(main):029:0> get 'table5', 'conx1:spaddress'
COLUMN              CELL
contextFmaily:time      timestamp=1381771496216, value= 11:35
contextFmaily:date      timestamp=1381771496216, value= 12/12/13
contextFmaily:sourceID      timestamp=1381771496216, value= spc12

hbase(main):029:0> get 'table5', 'conx1:sraddress'
COLUMN              CELL
contextFmaily:time      timestamp=1381771496216, value= 11:44
contextFmaily:date      timestamp=1381771496216, value= 12/12/13
contextFmaily:sourceID      timestamp=1381771496216, value= src12

hbase(main):029:0> get 'table5', 'conx1:srage'
COLUMN              CELL
contextFmaily:time      timestamp=1381771496216, value= 11:45
contextFmaily:date      timestamp=1381771496216, value= 12/12/13
contextFmaily:sourceID      timestamp=1381771496216, value= spc12
```

Figure 67: CASR Implementation in Hbase (Snapshot(2))

6.5 Summary

In this chapter, the implementation designs in Redis, MongoDB and Hbase NoSql databases are discussed. We described the structural characteristics of the three NoSql databases, their main features and limitations. Subject to each one's potential, we have developed mappings that respectively map the CASR generic model into Redis, MongoDB, and Hbase. These three NoSql databases have different features; therefore, each of them has a different implementation structure of CASR. Therefore, next chapter provides an evaluation for the three NoSql databases using different approaches. These include benchmarking, using experience and CASR requirements

Chapter 7

Testing and Analysis

In this chapter, we present a comparative study on the three NoSql implementations introduced in the previous chapter. We first examine the general abilities of NoSql databases in terms of its runtime and throughput using YCSB tool. Then, we provide a comparison between the general characteristics of the three NoSql databases. In the light of both the experimental results and the general characteristics, we analyse the proposed structure presented in the previous chapter for each NoSql database. Finally, we rank the NoSql databases based on their overall performance.

7.1 YCSB Benchmarking

Yahoo! Cloud Serving Benchmarking tool (YCSB) [CST+10] is designed to test the performance of NoSql databases without consideration to the structure of the database. The goal of this experiment is to examine the general runtime and throughput of the three NoSql databases with immense number of records. The runtime is the time necessary to execute operations on all methods, and the throughput is the number of operations performed in a second. Thus, the relationship between the runtime and throughput is a negative correlation. That is, the less the time necessary to finish the execution of operations on all records the more will be the number

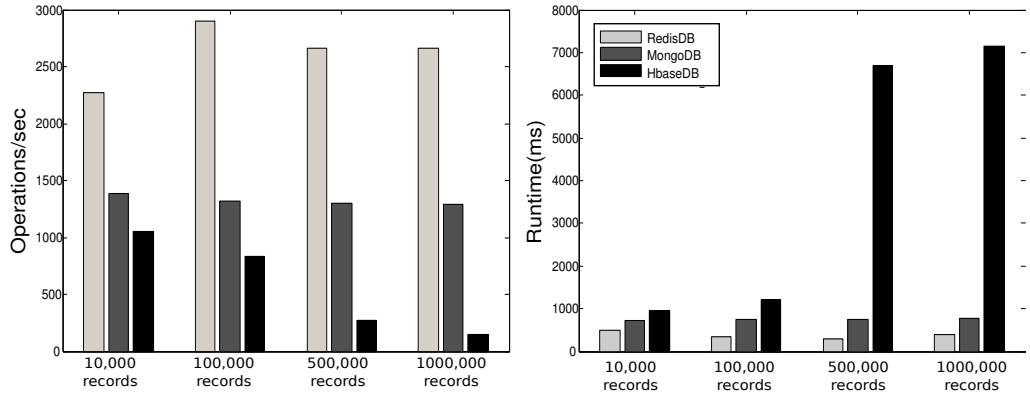


Figure 68: (A) Read/update ratio: 50/50

of operations performed per second. The basic database operations are: Insert, Retrieve, and Update. To test the possible combinations of those operations, YCSB provides six workloads. Each workload performs different combination of operations on different ratios. Table 4 shows the six workloads defined by YCSB. Also, each workload is executed with different number of records, 10,000, 100,000, 500,000, and 1000,000. We executed the operations three times for each workload and plotted the average of results in Figures (68, 69, 70, 71, 72, 73 and 74).

Workload	Operations Combination	Ratio
(A)	Read/Update	50:50
(B)	Read/update	95:5
(C)	Read/update	100:0
(D)	Read/update/insert	95:0:5
(E)	Scan/insert	95:5
(F)	Read/Read-Update	50/50

Table 4: The six workloads defined by YCSB

Remark 1 *For a fair examination, all experiments were performed on the same machine.*

From the experimental results we conclude that Redis occupies the first place in terms of runtime and throughput followed by MongoDB and Hbase is ranked last. However, YCSB does not consider complexity of structure. Therefore, the results could change dramatically with

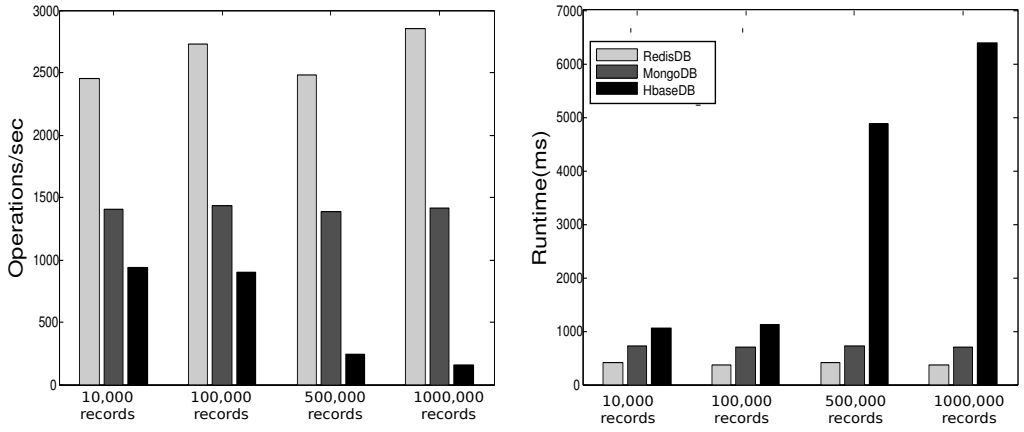


Figure 69: (B) Read/update ratio: 95/5

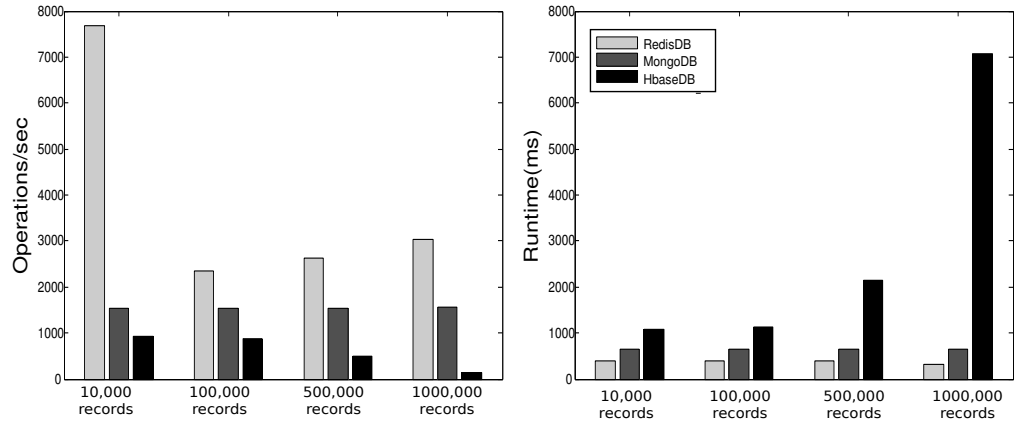


Figure 70: (C) Read/update ratio: 100/0

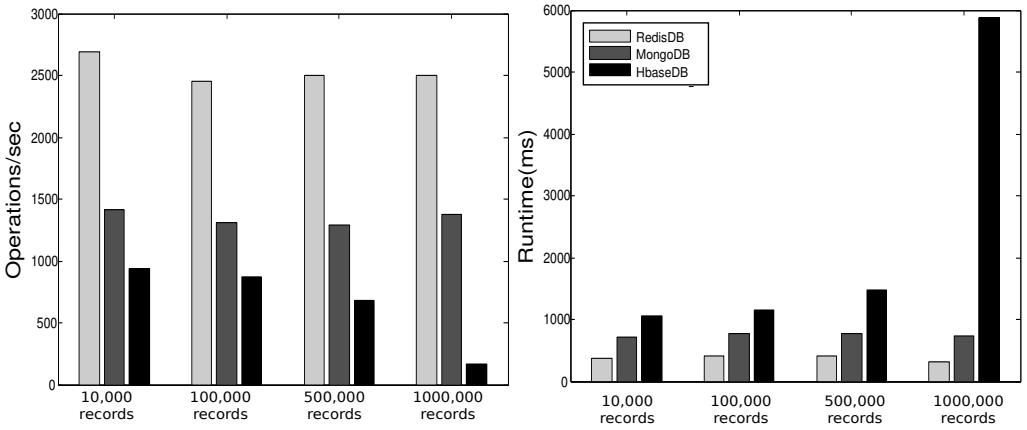


Figure 71: (D) Read/update/insert ratio: 95/0/5

more complex structures. Specifically, because Redis does not have pre-defined data structures, it consumes more operations to perform a single query. As a result, the overall Redis

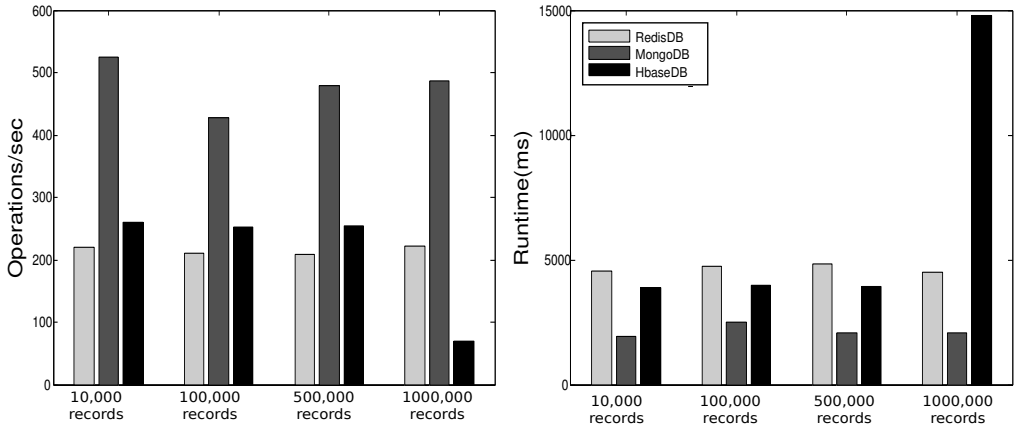


Figure 72: (E) Scan/insert ratio: 95/5

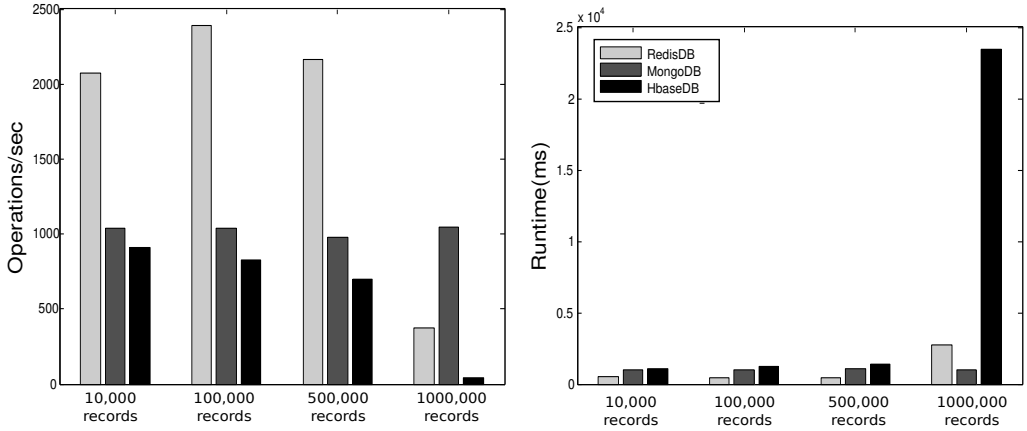


Figure 73: (F) Read/Read-Update ratio: 50/50

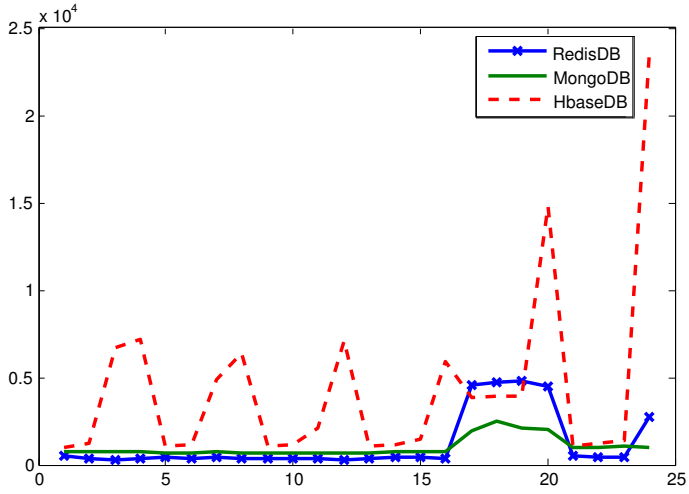


Figure 74: Overall Performance For all Workloads

performance may decrease, whereas MongoDB and Hbase may perform better with complex structures.

7.2 General NoSql Characteristics

In this section, we compare the general characteristics of each NoSql database and there structural features. The *CAP Theorem* [Bre00], which studied *consistency*, *availability*, and *partition tolerance* of NoSql databases, states that any NoSql database should have at least two out of three strong features. As NoSql databases are distributed, the partition tolerance support is mandatory which leaves the sacrifice on either consistency or availability. In [HHLD11], it is stated that MongoDB, Redis, and Hbase have the two strong features *consistency* and *partition tolerance* (CP). Redis is a flexible database but has some limitations, compared to the other two. The constraints on data type, indexing system, and key value structure make it more difficult to use with complex rich data. On the other hand, both MongoDB and Hbase can handle complex data. MongoDB supports hierarchical structures by permitting nested documents and allowing secondary indexing [Cho13]. In Hbase, hierarchical structures are supported by nested columns with multiple indexing [Geo11]. These features help developers to structure rich context data. MongoDB is easier than Hbase in configuring and coding. Table 5 illustrates this general comparison.

7.2.1 Redis

The primary strength of Redis is concentrated in its speed which is a common feature of key-value stores. Redis has high speed because it lives in memory and it supports replication which is useful for high read operations. Database developers could sacrifice speed to have durable database. Redis is durable database because its replication is built in master-slave methodology [RWC12]. In addition, Redis is powerful for its ability to store complex values by

	Redis	MongoDB	Hbase
CAP Theorem	CP	CP	CP
Strengths	High speed	flexibility,simplicity	versions support, compressions
Weaknesses	durability problem	difficult to update	can't work alone, or scale down
Maximum size	Chapter 6	16 MB	Cell 10-50MB
Indexing	one index	allows secondary index	cell queried by (row key, CF, CQ)
modeling ability	One structure, No hierarchy	Supports embedding	Tables and embedding

Table 5: A Comparison of the Structural Properties of Three Context Database Models

providing various data structures such as hash, sets and list. To manipulate data of these types Redis provides several operations such as Create, Retrieve, Update an Delete (CRUD), union and intersection. The union and intersection operations are useful to know, as an example, common friends between two users in social website. Moreover, Redis is easy to learn and use. The configuration is documented well and easy to understand. The shell command of Redis is clearly classified by their data type in their website [SN10]. Some other functionalities provided by Redis are useful in building databases such as key expiration. Key expiration is to have a key that can be used temporarily. This saves memory from storing unused keys and hence consistent performance. Key expiration time is assigned by a developer and hence its implicitly executed by the system.

As Redis is based on memory, data loss can happen when a shut down of the Redis server occurs. That is, if the shut down happened before writing the data to disk is complete, the loss might happen. Also, if clustering is needed in Redis databases developers should do it with clients, such as Ruby driver [RWC12], that support thess features. Also, Redis provides some data type that helps with structuring rich components, yet it is still not efficient enough for all complex structures. That is, one document in MongoDB or row in Hbase is actually represented by two or more records in Redis. Consequently, Redis will require many queries

to do the operations operations to retrieve one document or row, which could be done by one query in other databases. This requires some additional programming to trigger the several iterations on the stored records in Redis.

Through our experimental observations with Redis, structuring Redis without being aware of its limited data structures support can lead to serious performance issues. Even with careful considerations to its limitations, a single operation in complex structures can be translated to several operations when performed in Redis. This is because Redis does not support table or document structure. Therefore, there is a need to find a way to manually attach records by using key patterns and mappings. Yet, they increase the number of operations that are performed to retrieve one *ConfiguredService*. In Redis we only used hash and set data structures, since the ability to query each element of CASR registry is a necessity. Hash provides the ability to store fields names connected to their values. Deleting, adding, and updating specific fields can be performed with Hash. Sets can store different elements for a single value which is useful for some attribute and parameters of CASR data. However, other data structures such as list, is limited to the ability to remove only the first element of the list. Besides limited CRUD operations can be done on each element of the list. This limits our use to hash and set data to structure CASR registry.

7.2.2 MongoDB

MongoDB is a flexible database. It supports heterogeneous documents structures. That is, each document(row) in a collection(table) in MongoDB can have a structure different from other rows. Moreover, these structures can change dynamically. In addition, MongoDB manages massive data as it supports horizontal scaling and manages big number of requests by supporting master/slave replication [RWC12]. One of the most important features of MongoDB, is indexing which is not supported by Redis and Hbase. That is, aside from the mandatory indexing done by MongoDB system $\_id$, a secondary indexing is supported by MongoDB. MongDB

indices can come with a performance price which is worthy in front of the load that comes with querying data sets. That is, to get the power of indexing in MongoDB, the developer should define types of queries on the indexed data. If the data includes large data set and queried often, indexing it is worthy. Otherwise, it is not worthy to pay less performance price [RWC12].

MongoDB has some drawbacks associated with performance. MongoDB performs better in clusters with massive data as it is designed for large databases [Ban11]. This requires considerable management effort. Although embedding is a feature that enhances performance, it decreases querying flexibility. Information and fields that are inside the embedded document cannot be queried individually. However, the performance is enhanced as embedding helps in avoiding linking operations. That is, the embedded documents are usually sub-objects of the main objects that include related information. So, retrieving the main document will include sub-documents. Also, it is welcomed by MongoDB to retrieve the sub-document separately. The issue is when fields and data of these sub-documents are queried a lot. In this case embedding is not useful. Instead, these embedded documents can be stored in separate documents. In this case, linking between the main documents and the separated documents is required. This solves the querying problem but increases the overload that comes with retrieving the documents that includes a link to the sub-documents and hence performance degrades.

In using MongoDB we found that its concepts are easy to understand as it is close to traditional database. We did not have any issue to understand and implement database in MongoDB. The provided resources and online documentation are enough to start dealing with MongoDB. MongoDB features and characteristics help with structuring CASR registry. The ability to query data and fields easily meets our registry requirements in query system. Also, the ability to store different data type such as arrays and sub-document are very useful in structuring DKN, providers and *ConfiguredServices*. In addition, MongoDB supports dynamic document expansion which is essential feature to provide for CASR registry. CASR aims to provide SP to enhance their services by updating their service description adding new attributes and data. In Redis

and Hbase the dynamic expansion is supported, yet it is limited to the type of data that will be included in the records or documents. That is, if it is decided to expand records of SP by including a construct, such as `struct` which is commonly used in programming language, this can be easily performed with MongoDB. However, with Hbase and Redis there should be some analysis to find the proper structure to include those information within SP records, without affecting the performance and consistency.

7.2.3 Hbase

Hbase is featured with its ability to deal with massive data. That is, its performance peaks when storage includes huge amount of data locating in gigabytes or terabytes of memory [RWC12]. Hbase supports versioning and compression functionalities. These features omit the need to structure a database for history. Implicit versioning feature in Hbase [DKR13] gives the ability to get the history of data changes through a period of time. Hbase provides the functions to control number of versions and their durations. Also, Hbase has a semi-structured schema. This helps with structuring rich data. In addition, Hbase operations enable finding every single data stored in its tables. This features adds more expressive power to querying capabilities in Habse.

Although Hbase is a powerful system, there are some limitations with Hbase. Hbase supports *column families* in its structure. Each table can have more than one column family. However, Hbase performance degrades if the table has more than three CFs. This does not help when structuring rich components. If an application requires more than three CFs in a table, it can be constructed in Hbase but only a maximum of three CFs should be queried at a time in order to maintain the high level of performance. This is because each column family is stored in one region, and flushing and compaction operations are performed on the data to be retrieved from CFs. When several flushing and compaction operations are done for each query on different regions needless input/output operations become necessary, which in turn will increase

the system load. It is remarked in [RWC12] that

> *Hbase was quite challenge for us. The terminology can be deceptively reassuring, and the installation and configuration are not for the faint of heart.*

Indeed, we found that understanding and dealing with Hbase was not easy. There was much time spent to understand how to structure tables in Hbase, how to configure it on machine and what structural characteristics might lead to maximize Hbase potential.

7.3 Overall Verdict

Based on our observations, from the starting point of understanding the three NoSql, through to the final implementation, and in the light of the previously discussed experimental and structural comparisons, we summarize our own comparisons between the three databases. Also, we rank databases with respect to the suitability to our CASR registry. The following features are the most prioritized for our CASR registry. These features are briefly described as follows:

- Stable Performance: with different number of records, the performance shows stable behaviour.
- Indexing: the ability to index some fields of the records or the document.
- Fields Querying: the ability to query specific fields of the record or the document.
- Hierarchical Structure: to support embedding or linking among entities.
- Usage easiness: the ease of interacting with the NoSql database that includes configuration, installation and coding.

	Redis	MongoDB	Hbase
Stable Performance	Yes	Yes	No
Indexing	No	Yes	No
Fields Querying	Partial	Partial	Yes
Hierarchical Structure	No	Yes	Partial
Usage easiness	Yes	Yes	Partial

Table 6: Ranking Each NoSql database basing on CASR Requirement

Table 6 ranks each NoSql database in the light of the above comparison. Based on this ranking we recommend MongoDB as the most suitable NoSQL database for implementing CASR registry.

From the table, MongoDB is the database that meets most of the requirements that CASR registry needs. Even with the querying abilities, as it is mentioned before, it only comes with embedded documents and there are methods and solutions provided to overcome such a limitations. Having said that, Redis and Hbase have great features that can match other projects requirements.

7.4 Summary

In this chapter we introduced analysis on the three NoSql databases from the three perspectives general performance, general characteristics and personal observations. Then, we concluded our analysis with a summary that puts together all the three NoSql databases. Finally, we ranked the databases based on the ability to use them in our CASR registry.

Chapter 8

Conclusion

In this thesis, we proposed a structure and provided an implementation for a context-aware service registry (CASR). This registry is capable of storing and managing rich services, which have complex features, contracts, legal rules, and non-functional specifications. We analysed thoroughly the requirements of such a registry and introduced a generic structure for it. Also, we investigated in depth three NoSql database approaches and analysed their features, strengths, limitations, and structures. For each NoSql systems, we introduced a model for context-aware rich service definition. Finally, we implemented the three models and provided benchmark thorough analysis of the capabilities of each approach. We compared the results and outlined our analysis.

The contributions of this thesis are expected to have a positive impact and improve service provision and discovery. Technological services have become essential parts of daily life of people. They are provided in many essential sectors such as Health, Education, Entertainment, Business, etc. Including context in such service makes it smarter and well adapted to the needs of people. The context and history modeling parts of this thesis have been accepted for publication in *The Second International Conference on Context-Aware Systems and Applications*

(ICCASA) [AADDA13]. Providing an approach for modeling context-aware services and structuring and implementing context-aware registries brings a significant contribution to the field of service computing.

8.1 Future Work

In order to reach its full potential, CASR has to be included in a bigger framework that supports context and trustworthiness. Ibrahim [Ibr12] has proposed a framework for trustworthy context-aware service publication, discovery and delivery. This is shown in Figure 75. We have explained details of the design for the context, *ConfiguredService* and service registry. In this section, we explain how our service registry model will fulfil the goals of the framework architecture for service publication and service discovery. In particular, we want to emphasise the design components that should be constructed in future.

The three goals stated by Ibrahim [Ibr12] for designing service registry are:

- It should be a simplified browsing media for SRs and an efficient publication media for SPs.
- It should provide secure access for SPs and SRs.
- It has build in semantics for publishing functionalities under different domains.

Figure 76 shows the components that are required to provide a full implementation of CASR. We briefly describe these components and their interactions with CASR in order to fulfil the three goals that are stated above.

The structure of CS storage, which is the main registry for services, has been thoroughly explained in previous sections. We assume that there exists a Trust Authority (TAU) which has the export knowledge, skills and resources to construct domains and sub-domains and manage identities of service providers and requesters using Role-Based Access Security (RBAC). Initially,

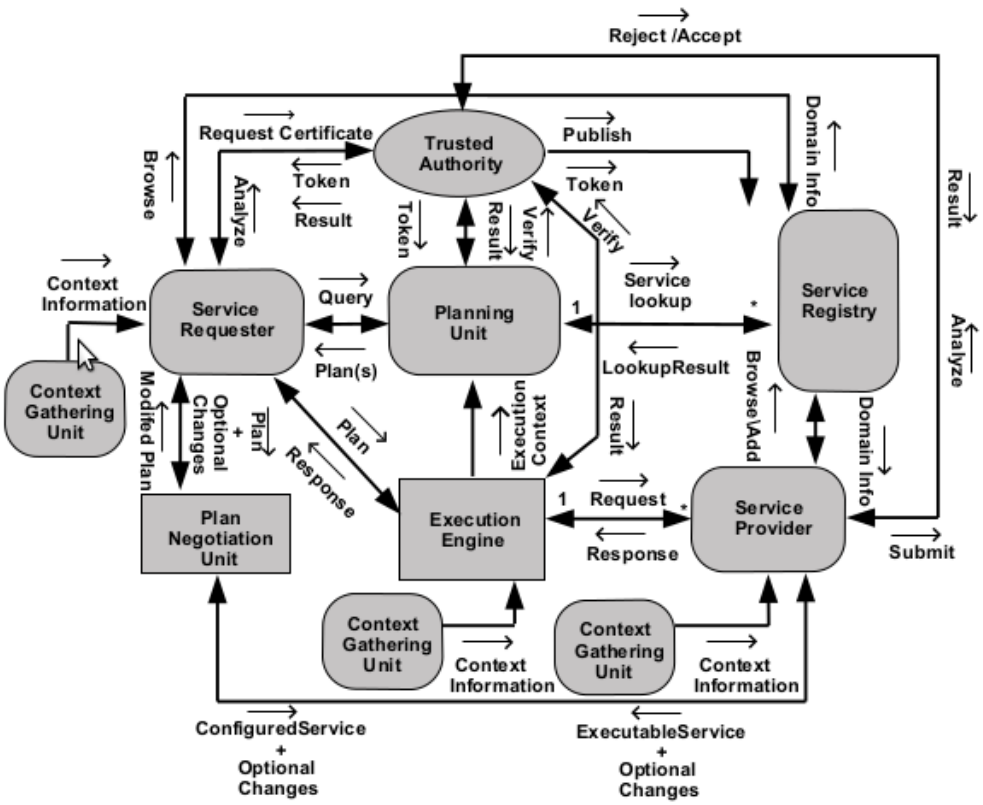


Figure 75: The FrSeC Framework Architecture [Ibr12]

the TAU will identify the domains of interest and the sub-domains within each domain. For example, Education, Transportation, HealthCare and Entertainment are domains, with each domain the sub-domains are identified and this information is structured as skeleton within the CS storage. This skeleton will be consistent the service registry model that we discussed earlier. Functionalities under each sub-domain will be added as and when service providers upload their services in the CS storage. This process of uploading is shown in Figure 77.

Figure 77 shows the sequence of SP actions. In step (1), the SP is to be registered by the TAU, this step is required to fulfil the secure access goal. After the is successfully registered, the SP is assigned a token by TAU, who also forwards the token with the identity of SP to be saved in Tokens storage. This is shown in Figure 77 as step (2). In steps (3) and (4) the confirmation and acceptance of SP in the system happen. The token has time window specifying the session

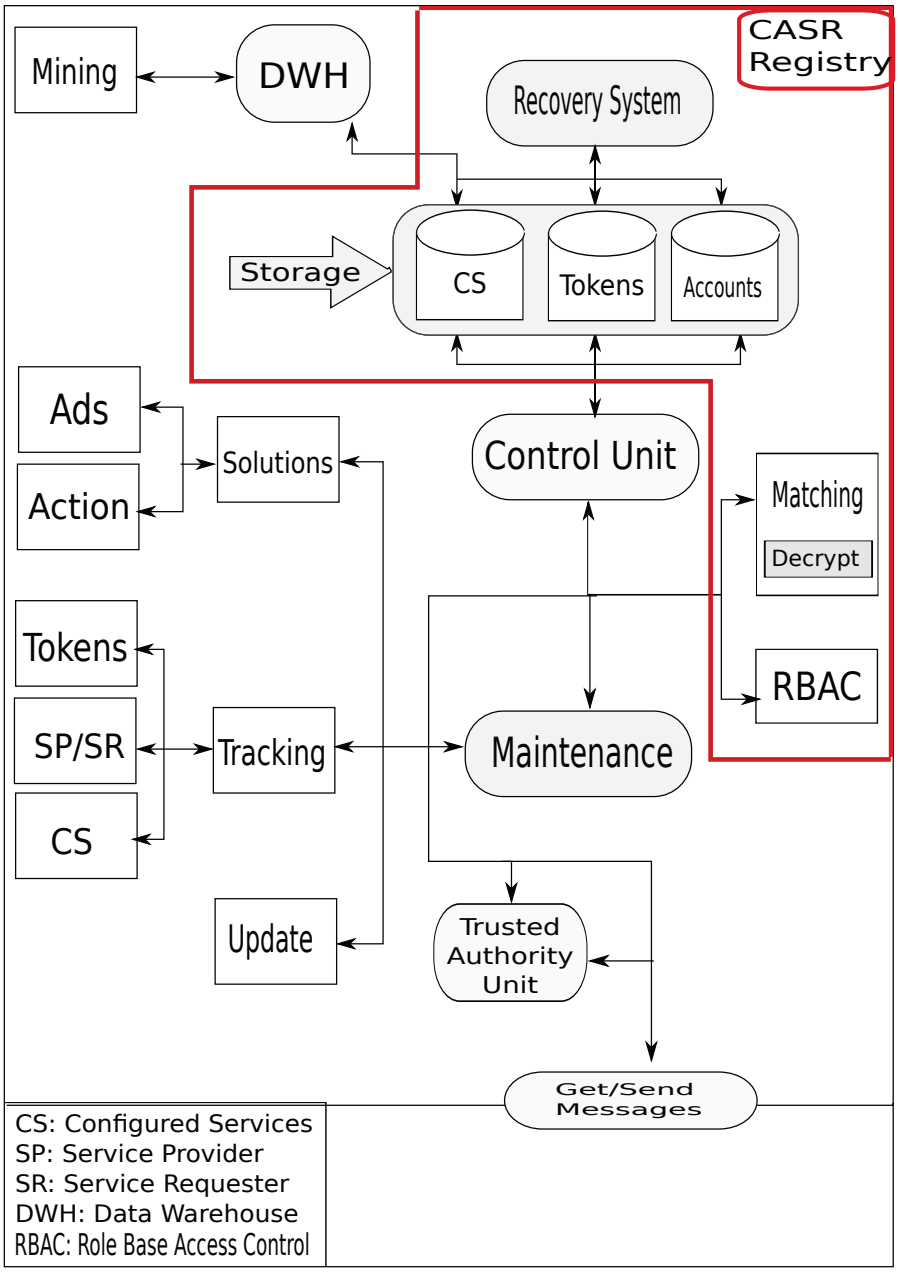


Figure 76: The main structure Service Registry

duration allocated for the SP. Within this time the SP expected to complete the rest of the action in Figure 77. In case the time expires before this is accomplished, the whole scenario shown in Figure 77 has to be repeated.

The semantics of the rest of the steps are as follows:

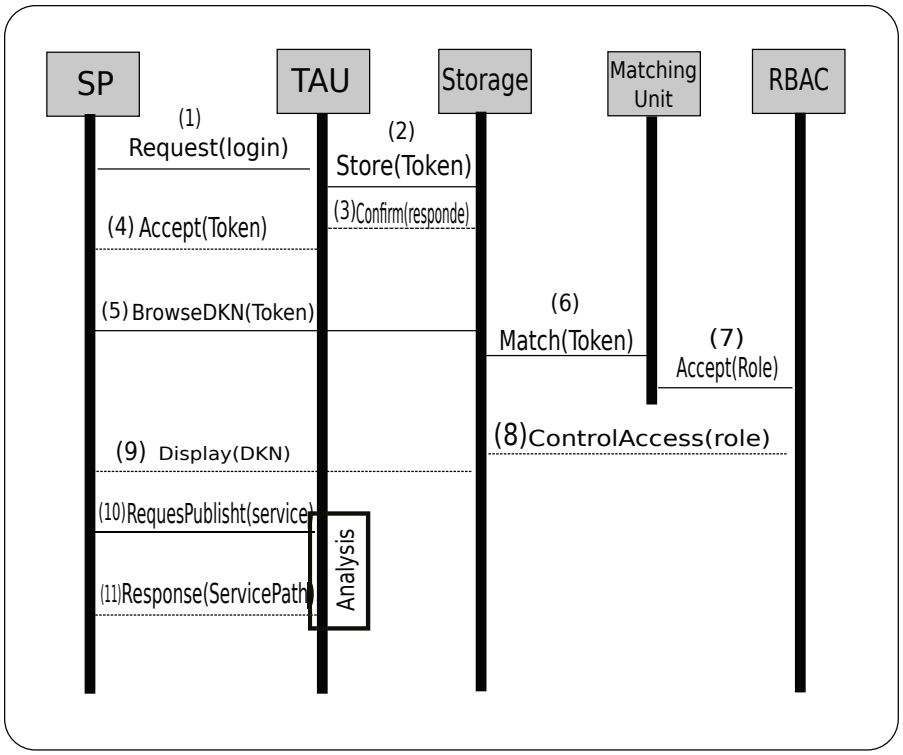


Figure 77: Service Publication Sequence Diagrams

- (5) Now after the SP is accepted to access the storage, SP request Browsing DKN sending its token with the request.

- (6) Then, the The storage takes this token and send it to the Matching Unit (MU) to first match it with the one stored in the Token storage and if it mutual, the role is sent to RBAC. The reason for matching the tokens is to support security goal. The information included in token sent by SP could be changed or faked by any reason. Therefore, the system stores a copy to authorize the identity of the sender and hence secure the system.

- (7) The RBAC takes this role and checks the domains, sub-domains and functions that are eligible for this SP. If the role of SP exists in the ProviderRole list that is attached with every domain, sub-domain and function, then they are eligible to be displayed. The RBAC takes the list of eligible domains, sub-domain, and function giving it to storage. (8) the storage takes this list and display it to SP.

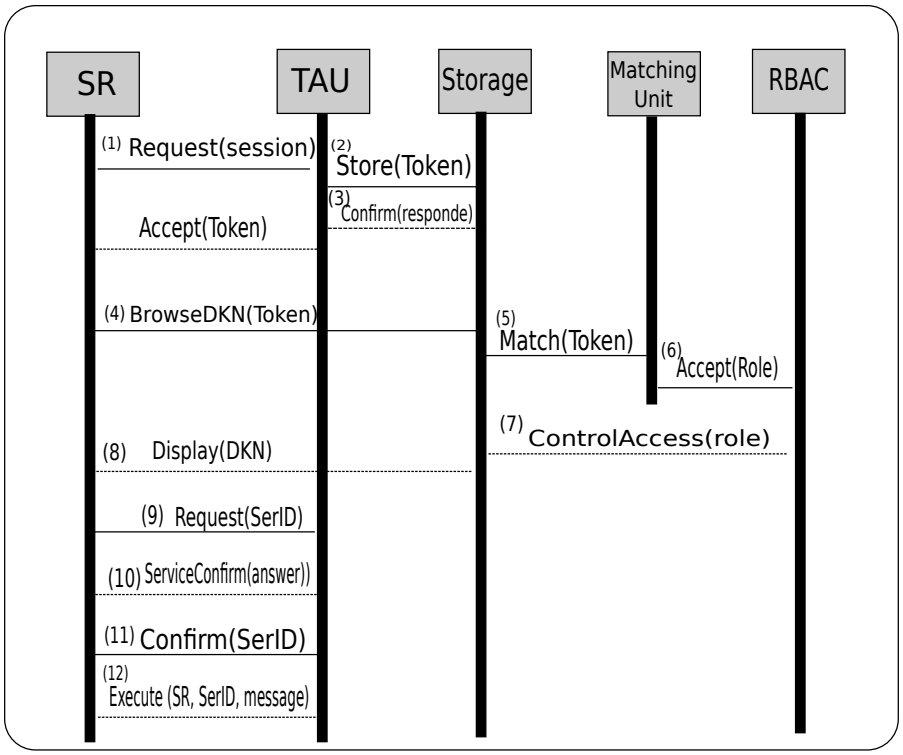


Figure 78: Service Discovery Sequence Diagrams

- (10) SP select the proper path for its service, fills the service publication request and sends the request. (11) The TAU analyses the request and makes a decision. If it is accepted then a confirmation message is sent to SP including other information attached with this decision such as contract renewal date. If the publication is not accepted, the TAU sends refusal letter that includes reasons for this refusal that could help SP to avoid those issues when reordering publishing the service.

For the service requester SR follows the steps in Figure 78 to discover services. The semantics of the steps are similar to the steps are shown in Figure 77. However, (10),(11) and (12) are actually steps for requesting a service, then TAU requests confirmation from SP and, finally, after SR confirms to request the services, service execution process starts.

The rest of the units shown in Figure 76 are necessary to manage the system evolution. The functionality of these units are explained below.

The Control Unit (CU) is the unit that control input and output in storages and from storages. That is, this unit works as an interface of storages to allow or not to allow communication between storages and other units. It is needed to help processing other functionalities such as service publication. That is, the final step for service publication is to accept the request coming from SP by TAU. When TAU accepts the request, the service information is sent to service registry to be stored, the request goes through the CU to validate request and sends it to the storage and to refuse request in case of data loss during request transferring.

Maintenance Unit (MU) that is responsible for service development and improvement. This is done by requesting frequent update and checking on the stored data. Service improvement is done by two units one is tracking and another is solution. The tracking unit tracks the least used or unused *ConfiguredServices* in the system. These *ConfiguredServices* can be improved by providing some solutions. Therefore, the *ConfiguredServices* with low usage are sent to solution unit to provide a development plan to improve the service usage. These plan includes actions such as offers and deals provision, service compositions with the help of Action unit, or advertising with the help of Ads unit. The Action unit could make firm decisions such as deleting the *ConfiguredService* , if the providers of those *ConfiguredServices* do not conform to the rules of the agreement that are provided to service providers when they start working with CASR registry. The tracking unit also tracks service provider and service requester accounts and tokens expiry dates to command actions in some particular cases. These cases and actions should be defined and documented in the CASR registry.

The Data warehouse Unit is to store all transactions done by the registries which help to provide long term process report of each service that helps SPs to know many aspects of where they can improve their services.

Bibliography

[AADDA13] Alaa Alsaig, Ammar Alsaig, Mubarak Dr.Mohammad, and Vangalur Dr. Alagar. Storing and managing context and context history. In *The Second International Conference on Context-Aware Systems and Applications*, Phu Quoc, Vietnam, November 2013.

[AS96] Varol Akman and Mehmet Surav. Steps toward formalizing context. *AI magazine*, 17(3):55, 1996.

[Ban11] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011.

[BBH+10] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.

[BCE+02] Tom Bellwood, Luc Clément, David Ehnebuske, Andrew Hately, Maryann Hondo, Yin Leng Husband, Karsten Januszewski, Sam Lee, Barbara McKee, Joel Munter, Claus von Riegen, and SAP. Uddi spec technical committee specification. *URL:http://uddi.org/pubs/uddi-v3.00-published-20020719.htm*, 2002.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[Cat11] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[CC81] Herbert H Clark and Thomas B Carlson. Context for comprehension. *Attention and performance IX*, pages 313–330, 1981.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. *URL:http://www.w3.org/TR/wsdl*, 2001.

[CDG+08] 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. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[Cho13] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly, 2013.

[CLC10] Lu Chen, Yan Li, and Randy Chow. Enhancing web service registries with semantics and context information. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 641–644. IEEE, 2010.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[DAS01] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, 2001.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[DKR13] Nick Dimiduk, Amandeep Khurana, and Mark Henry Ryan. *HBase in Action*. Manning, 2013.

[Geo11] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[GS07] Vincenzo Grassi and Andrea Sindico. Towards model driven design of service-based context-aware applications. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, pages 69–74. ACM, 2007.

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[HL06] Yan Ha and Roger Lee. Integration of semantic web service and component-based development for e-business environment. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 315–323. IEEE, 2006.

[Ibr12] Naseem Ibrahim. Specification, composition and provision of trustworthy context-dependent services. Technical report, Concorida Univeristy, 2012.

[K+05] LF Kenney et al. Soa registries and policy enforcement bolster soa governance and consumption. *Gartner Research*, 2005.

[Kei08] Jones Keith. Building a context-aware service architecture. *URL:http://www.ibm.com/developerworks/architecture/library/ar-conawserv/index.html*, 2008.

[Lea10] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[Lee08] Youngkon Lee. Quality context composition for management of soa quality. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 330–335. IEEE, 2008.

[LGZ+05] Jiamao Liu, Ning Gu, Yuwei Zong, Zhigang Ding, Shaohua Zhang, and Quan Zhang. Service registration and discovery in a domain-oriented uddi registry. In *Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on*, pages 276–283. IEEE, 2005.

[LL09] Ruiming Li and Nianlong Luo. xuddi: A framework for uddi registry based on purexml. In *Information Processing, 2009. APCIP 2009. Asia-Pacific Conference on*, volume 1, pages 510–513. IEEE, 2009.

[Min08] Alexander Mintchev. Interoperability among service registry implementations: Is uddi standard enough? In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 724–731. IEEE, 2008.

[ML90] John McCarthy and Vladimir Lifschitz. *Formalizing Commonsense: Papers by John McCarthy*. Greenwood Publishing Group Inc., 1990.

[Rud56] Carnap Rudolf. Meaning and necessity, 1956.

[RWC12] Eric Redmond, Jim R Wilson, and Jacquelyn Carter. *Seven databases in seven weeks: A Guide to modern databases and the NoSQL movement*. Pragmatic Bookshelf, 2012.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE, 1994.

[SB05] Quan Z Sheng and Boualem Benatallah. Contextuml: a uml-based modeling language for model-driven development of context-aware web services. In *Mobile Business, 2005. ICMB 2005. International Conference on*, pages 206–212. IEEE, 2005.

[SCA06] CM Saracca, Don Chamberlin, and Rav Ahuja. Db2 9: pure xml–overview and fast start. *IBM Redbooks*, 2006.

[SN10] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2010.

[Tay10] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(Suppl 12):S1, 2010.

[TJ10] Rob Tweed and George James. A universal nosql engine, using a tried and tested technology, 2010.

[TKS+10] Martin Treiber, Kyriakos Kritikos, Daniel Schall, Schahram Dustdar, and Dimitris Plexousakis. Modeling context-aware and socially-enriched mashups. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, page 2. ACM, 2010.

[Wan06] Kaiyu Wan. *Lucx: Lucid enriched with context*. PhD thesis, Concordia University, 2006.

[web] Uddi tutorial. *URL:http://www.tutorialspoint.com/uddi/index.htm*.