UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANDERSON DIDONÉ FOSCARINI

Development of a Context Broker and High Availability resource proposal

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Taisy Silva Weber
Coadvisor: M.Sc. Marcos Rates Crippa

Porto Alegre
July 2015

“The true sign of intelligence is not knowledge,
but imagination.”
— ALBERT EINSTEIN

ACKNOWLEDGEMENTS

I would like to express my very great appreciation to both my advisors, for the patience and kindness on the journey of getting this work done.

Special thanks to my parents, for all the support they gave me throughout the years, always providing the best conditions for my living. I surely know how lucky I am, and I value it every day. I wouldn't be here if it wasn't for you.

I am particularly grateful for the good moments and the assistance given by my brother during the years, both computer and life related, specially in the final days of this work.

I must save a little space to thank all my friends, the old and the new ones, who made the last 7 years of my life very pleasant. A special thanks to those who picked up on me so I would focus and finish this work.

I would like to thank TU Kaiserslautern for the opportunity given and specially the Informatics Institute from UFRGS, for the excellence on learning and research, being the best computer science course in the country for many years, and having international recognition. This is where I met great professors from whom I learned not only the theory and application of computer science, but also lessons in life. It was surely a great honor to grow both academically and personally inside this great atmosphere, and if I decide to continue my studies, I will strongly consider choosing this institution.

And finally thank you, for reading this. I hope you have a good time.

ABSTRACT

Context is everywhere. From a mobile phone in a person's pocket to a temperature sensor in the middle of a forest. Context data can be used to help determine the importance of information and services on an environment, and decide whether to make them available to users or not, and how to process them. Applications that use context are called context-aware. With today's large offer of sensing technologies, and the presence of various kinds of sensors in every mobile phone, it has become easier and more useful to sense context in several situations. The use of context data has become a very powerful tool of personalizing interaction with users and the behavior of systems. By improving the computer's access to context, the richness of communication in human-computer interaction and the presence of more useful computational services can be increased. Context-aware computing is a new and broad area of research. Although its wide range of applicability, not many efforts are made towards specific solutions, including ones related to dependability and fault tolerance. This work's main objective is to develop a Context Broker following a previous definition, on a different platform, and to propose a way to add a resource that gives the Broker high availability characteristics, taking a first step towards the development of a Dependable Context Broker.

Keywords: High Availability. Context broker. Context computing. Fault tolerance.

Desenvolvimento de Broker de Contexto e proposta de recurso de Alta Disponibilidade

RESUMO

Contexto está em todo lugar. De um telefone celular no bolso de uma pessoa à um sensor de temperatura no meio de uma floresta. Dados de contexto podem ser usados para auxiliar na definição da importância de informações e serviços em um ambiente, e decidir se as torna disponíveis ao usuário ou não, e como pode processá-las. Aplicações que usam contexto são chamadas sensíveis ao contexto. Com a atual grande oferta de tecnologias sensoriais, e a presença de vários tipos de sensores em cada telefone celular, têm se tornado mais fácil e prático captar contexto em diversas situações. O uso de informações de contexto se tornou uma ferramenta poderosa para personalizar a interação com usuários e o comportamento de sistemas. Ao melhorar-se o acesso de um computador ao contexto, aumenta-se a riqueza das comunicações em interações humano-computador e a presença de serviços computacionais mais úteis.
Computação sensível a contexto é uma nova e ampla área de pesquisa. Não são muitos os esforços em soluções específicas, inclusive as relacionadas à área de tolerância a falhas e confiabilidade. Este trabalho tem como objetivo desenvolver um Broker de Contexto e propor a adição e recursos que o dêem características de alta disponibilidade, dando um primeiro passo em direção ao desenvolvimento de um Broker de Contexto Confiável (Dependable).

Palavras-chave: Alta disponibilidade. Broker de contexto. Computação de contexto. Tolerância a falhas.

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

CxB	Context Broker
CxC	Context Consumer
CxP	Context Provider
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
SOA	Service-Oriented Architecture
TU-KL	Technische Universität Kaiserslautern
UFRGS	Universidade Federal do Rio Grande do Sul

CONTENTS

1 INTRODUCTION

Humans can interact easily, and naturally gather information about the context they're in. This is due to various factors as language, common understanding of the world's behavior and everyday situations, and the ability to understand implicit symbolization and effects, based on common knowledge. Unfortunately, a computer does not have this capability: it needs to be guided, to be told what to look for, what to sense, and how to interpret it; it needs explicitness (DEY, 2000).

Gathering context information is a way of computers to interact with its surroundings, collecting information about the user and the environment he's inserted. With today's wireless communications technologies, mobile and ubiquitous computing, sensors, etc., the information can be collected silently, i.e., without the need of a user to explicitly input it, and is more dynamic, as it rapidly changes not only through user interaction, but also when the context changes itself. An example is given below.

A context-aware system could be used in an intelligent store. Let's assume we have information from a client's location, using its mobile phone position, and information of a store location and products, registered in the context-aware system as the store's **context**.

This information can be combined to show in the client's mobile phone offers and products from the store, either when the client passes by or enters it. If the client register its personal data to the context-aware system, it can even suggest products that would interest the client, resulting in a context-directed advertisement. Going further in this idea, the client could also see what products of his size are available at the store, if these context information about the client are registered in the context-aware system. This was a mere example, there are many other ways of using context data in real-world applications, as tourist context-aware recommendation content, ebooks interactions, content share services, etc. (MOLTCHANOV et al., 2011).

A context-aware system usually consists of Providers and Consumers of context information, forming a Service-Oriented Architecture (SOA). When this system grows, the use of a Broker is recommended, centering the message exchanging in the Broker, letting Providers and Consumers be simple systems and avoiding overload among them. Among many extensions that can be made to a Broker application, high availability comes out as a very interesting one. Given today's highly competitive perception of market, the availability of a solution can be decisive in the satisfaction rating of its services.

Motivation: in the year 2014 I was given the opportunity to be part of an exchange program between Universidade Federal do Rio Grande do Sul (UFRGS) and Technische Universität Kaiserslautern (TU-KL) in the city of Kaiserslautern, Germany. I was selected to work on the Wicon Research Group (TUKL, 2015), under the supervision of Msc. Marcos Rates Crippa, the co-advisor of this work. Initially I was given tasks of documenting and learning about Context and the Context Broker solution he had developed previously within TU-KL. As I was approaching the end of my computer science course at UFRGS and needed a final project subject, Marcos presented to me some options, among them the study of a method to add high availability technique to the Broker architecture. Then, I decided to develop my own Context Broker solution, following the same definitions, but in a different programming language, so it would arise as a challenge for me, and later study for a protocol to seek high availability in the Broker system.

This work follows a previous work done at UFRGS in cooperation with TU-KL (CRIPPA, 2010). In that work, a Context Broker was defined and created using Java. The goals of this work are to create a regular Context Broker following previously defined architecture, in a different programming language, but that has the same behavior from the client point of view, showing that two different solutions can work side by side, without the need of modifying the client. This work also aims at demonstrating the feasibility of the construction of a highly available Context Broker, as an extension of the regular one.

The Context Broker structure presented in (CRIPPA, 2010) and in this work is still a new approach to context-aware systems. As far as I've searched, I've found no production regarding high availability or any other dependability approach to a Context Broker in the literature. What exist are broker-based tools that help a system become highly available (MAFFEIS; SCHMIDT, 1997) (NATARAJAN et al., 2000).

The text is organized as follows: Chapter 2 is divided on definitions of Context and Fault Tolerance terms, with Section 2.1 focusing on the former and Section 2.2 on the latter. Chapter 3 presents the design and implementation of the regular Broker, and the strategies to incorporate High Availability to it. Chapter 4 shows the tests made in this work, and finally Chapter 5 brings the conclusions and future work.

2 DEFINITIONS

This work proposes the application of a high availability technique to a context broker system. For a better understanding of the system and its development, definitions of Context, Context-Aware System, Context Representation, Fault Tolerance and High Availability related terms are presented.

2.1 Context

Context has had many definitions throughout the years. The first definition of Context regarding human-computer interaction related to location, identities of nearby people and objects, and the changes happening to those (SCHILIT; THEIMER, 1994). Similarly, a later definition sees context as location, people around the user, time of day, season, temperature, etc. (BROWN; BOVEY; CHEN, 1997). Many authors have also defined context using synonyms, the idea of "environment" for example, what the computer knows about the user's environment (BROWN, 1995), or context as user's situation (FRANKLIN; FLASCHBART, 1998). Thus, a lot of definitions existed, but they all ended up being too specific. Context is about the whole situation of an application and its users, and we can't really define it as being too specific to something like a location, or the environment a user is in.

Therefore, looking for a broader definition of context, this work uses the following: "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, including the user and application themselves." (DEY, 2000).

2.1.1 Context-Aware System

Just as context, context-awareness has had several definitions over the years. The first definition restricted it to applications informed about context and applications that adapt themselves to context (SCHILIT; THEIMER, 1994). Later on, synonyms have been used to define a context-aware system: reactive (COOPERSTOCK et al., 1995), responsive (ELROD et al., 1993), situated (HULL; NEAVES; BEDFORD-ROBERTS, 1997), context-sensitive (REKIMOTO; AYATSUKA; HAYASHI, 1998) and environment-directed (FICKAS; KORTUEM; SEGALL, 1997). All these definitions refer to either using or adapting to context, however, a more global definition is needed, covering every interaction with context made by the system.

The definition used in this work aims to be more embracing: “A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user’s task.” (DEY, 2000).

A context-aware communication system usually comprises several context management functionalities, where context acquisition and provision are the most important ones. We can divide the system in two main component types: context providers and context consumers; a combination of both is also possible. Given this structure, a small scale system could work with direct communication between context providers and consumers, but when a large scale system is built, network boundaries, mobility and other scaling factors give rise to the necessity of having assisting communication mechanisms, e.g. a broker between the consumers and providers (KIAN et al., 2010).

2.1.2 Broker

A context-aware system falls under the definition of a Service-Oriented Architecture (SOA), where one can find Consumers and Providers of context information, and they all provide services for each other. They can interact directly, but as systems get bigger and more complex, a brokering component is required.

A Broker is the mediator for the data exchanged by Providers and Consumers. An advantage of using a Broker instead of direct communication between the two other parts is that, one part can send only one message to the Broker, and then the Broker sends this information to many other parts, instead of the first part sending it to each other part. It allows the parts to be very simple systems, that won’t need to support multiple connections, e.g. sensors that don’t have a complex communication system, and can’t handle many connections, they are connected only to a Broker and the Broker is responsible of handling many connections to other parts of the system. A Broker application pattern, as seen on Figure 2.1, is based on a 1-to-N topology that separates distribution rules from the applications. It allows a single interaction from the source application to be distributed to multiple target applications concurrently, reducing the proliferation of point-to-point connections (ARSANJANI, 2004).

In such a way, a Context Broker is essential to the well-behavior of a large context-aware system.

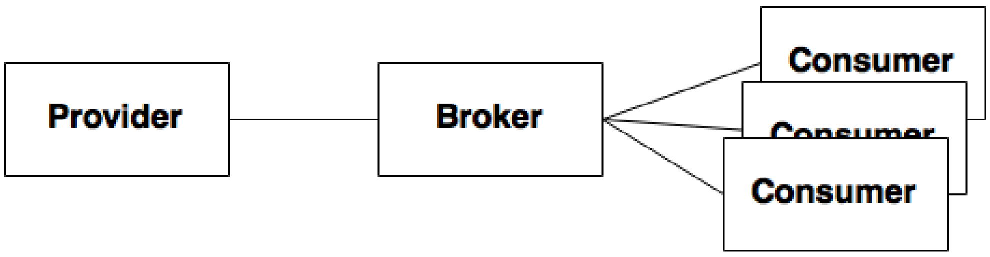


Figure 2.1 – Broker application pattern

2.1.3 How to represent Context

Context-aware applications deal with the who's, where's, when's and what's (the activities that are occurring) of entities, and interpret this information to define why a situation is occurring. Then, the designer of the application must decide what to do with the information. Once we have the information available, either through automated sensors or through user's interference, we need to represent it in a way a machine can process and store it (DEY, 2000).

Context can be modeled in many ways, the most relevant options being: key-value, markup scheme, graphical, object oriented, logic based and ontology based models (BALDAUF; DUSTDAR; ROSENBERG, 2007). As this work is developed following the same core as (CRIPPA, 2010), it uses the same context representation model: a markup scheme variation, **ContextML** (KNAPPMEYER et al., 2010). The network nature of the messages (HTTP messages, in this case) facilitates textual, non graphic model.

2.1.4 ContextML

ContextML is an XML-based representation schema for context information, where it is categorized into scopes and related to different types of entities. It is designed to be used with REST-based communication between the framework components (KNAPPMEYER et al., 2010). It was created within a project called C-CAST (Context Casting) (ICTGROUP, 2015), a collaborative work of many companies, research centers and universities, and its main objective is to evolve mobile multimedia multicasting to exploit the increasing integration of mobile devices with our everyday physical world and environment (CRIPPA, 2010). The architecture of the system presented in this work is based on the architecture of the C-CAST Project.

The system consists on three core components: **Context Provider**, **Context Consumer** and **Context Broker**. They use an idea of *entity* and *scope* to represent context information, and communicate through particular types of ContextML messages. Basic definitions of all these components are given below (KNAPPMEYER et al., 2010).

2.1.4.1 Context Provider

A **Context Provider** (CxP) provides context information of a certain type, e.g. weather, location, activity, etc. It gathers data from sensors, network, user interactions, or other sources. A CxP is specialized in a specific domain of context information (location, weather etc).

2.1.4.2 Context Consumer

A **Context Consumer** (CxC) queries for and uses context data, therefore is a context-aware application. A CxC can retrieve context information asynchronously through a subscription method, or by a synchronous method where it requests the Broker for a specific information or for a particular Provider interface, to query the Provider directly.

2.1.4.3 Context Broker

A **Context Broker** (CxB) is the central component of the architecture, and is the focus of this work. It handles and aggregates context information, and is an interface between the other architecture components. The CxB allows CxCs to subscribe to context information, and CxPs to provide this information. It also provides a lookup service, where the CxCs can query the CxB for CxPs that have a particular capability, depending on the CxC's interest.

2.1.4.4 Entity and Scope

An entity is the subject of interest which context data refers to, and it is composed of two parts: a type and an identifier. The type refers to the category of the entity: username for human users, imei for mobile devices, room for a room with sensors, etc. The identifier specifies a particular item within a set of entities of the same type.

A scope is a set of closely related context parameters. Every context parameter has a name and belongs to only one scope. The parameters of a scope can only be requested, updated, provided and stored at the same time (an atomic operation), making the data always consistent. For example, a scope *position* has latitude, longitude and accuracy attributes; any operation on this scope is performed on all these attributes: if the latitude is updated, so is the longitude and accuracy, what is correct, because otherwise it would not make sense. Entity-scope association is illustrated in Figure 2.2.

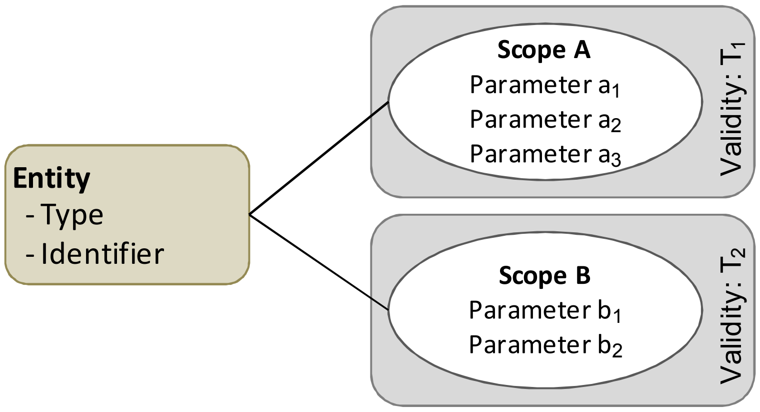


Figure 2.2 – Entity and Scope relationship
(KNAPPMEYER et al., 2010)

2.1.4.5 ContextML Messages

Within the architecture of the system, context is registered, updated and queried following a set of pre-defined ContextML messages (KNAPPMEYER et al., 2010). The ContextML message types used in this work and their usage are shown below, as well as a simple example of each one.

Advertisement Message

An Advertisement Message is used by the Context Provider to register its capabilities to the broker. It informs the CxP's access url (*urlRoot*), what scopes it supports (*scopes*), its identifier (*id*), and optional information about the CxP's location. An example of a Advertisement Message can be seen in Figure 2.3.

CxP Lookup Message

When a CxC wants to know where it can find a specific scope, it can query the CxB about which of the registered Providers has the desired information. The Broker replies with a ContextML message, describing the Providers that match with the data required by the CxC. An example can be seen in Figure 2.4.

```
<contextML>
    <ctxAdvs>
        <ctxAdv>
            <contextProvider id="PersonalDataProvider" v="1.0"/>
            <urlRoot>http://foscarini.biz/PersonalDataProvider</urlRoot>
            <providerLocation>
                <lat>-30.0142748</lat>
                <lon>-51.1664642</lon>
                <location>Umbu Street, 265, Porto Alegre, Brazil</location>
            </providerLocation>
            <scopes>
                <scopeDef n="attributes">
                    <urlPath>/getAttributes</urlPath>
                    <entityTypes>username</entityTypes>
                    <inputDef>
                         <inputEl name="height" type="float"></inputEl>
                         <inputEl name="weight" type="float"></inputEl>
                         <inputEl name="shoeSize" type="float"></inputEl>
                     </inputDef>
                 </scopeDef>
             </scopes>
         </ctxAdv>
     </ctxAdvs>
 </contextML>
```

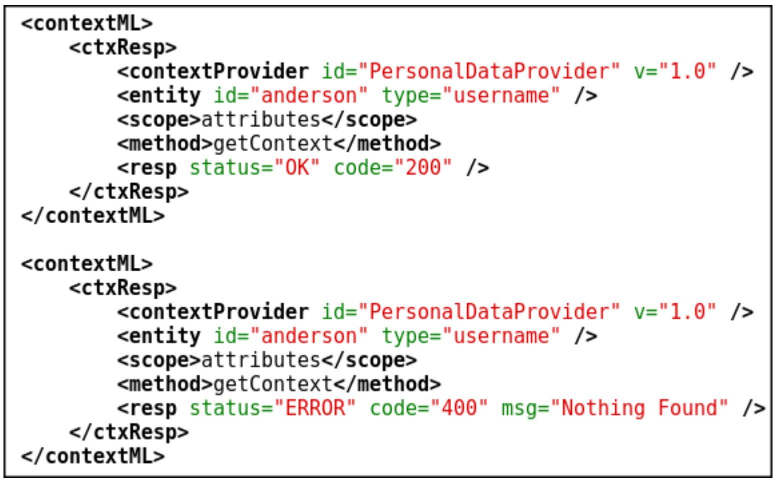
Figure 2.3 – Provider Advertisement example message

```
<contextML>
    <ctxPrvEls>
        <ctxPrvEl>
            <par n="scope">attributes</par>
            <parA n="contextProviders">
                <parS n="contextProvider">
                    <par n="id">PersonalDataProvider</par>
                    <par n="url">http://foscarini.biz/PersonalDataProvider</par>
                </parS>
            </parA>
        </ctxPrEl>
    </ctxPrvEls>
</contextML>
```

Figure 2.4 – Providers Lookup example message

ACK Message

Acknowledgement is a control message that confirms the execution of various management actions (e.g. advertisement, context update). Each ACK message contains the status of the operation, the HTTP response code, and the identification of the method it corresponds. It also has optional fields to inform scope and entity information. An example is shown in Figure 2.5.



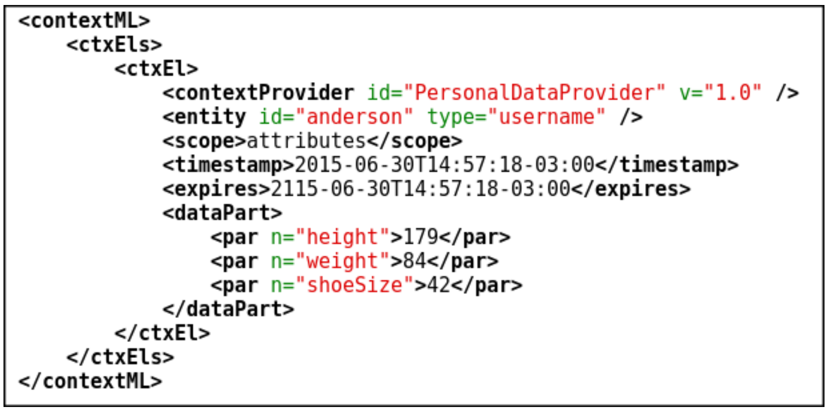
```
<contextML>
    <ctxResp>
        <contextProvider id="PersonalDataProvider" v="1.0" />
        <entity id="anderson" type="username" />
        <scope>attributes</scope>
        <method>getContext</method>
        <resp status="OK" code="200" />
    </ctxResp>
</contextML>

<contextML>
    <ctxResp>
        <contextProvider id="PersonalDataProvider" v="1.0" />
        <entity id="anderson" type="username" />
        <scope>attributes</scope>
        <method>getContext</method>
        <resp status="ERROR" code="400" msg="Nothing Found" />
    </ctxResp>
</contextML>
```

Figure 2.5 – ACK and NACK example messages

Context Representation Message

This is the form of representing context data in the architecture. When a Consumer requests or subscribes to a context scope, it receives a ContextML message with the element *ctxEl*, when the information queried is available. *ctxEl* contains information of the provider that has the context queried (*contextProvider*), the entity and scope it is related to, and the context data in the *dataPart* element. *par*, *parS* and *parA* are constructors to store name-value pairs and attribute collections (structs and arrays) respectively. Every context information that is exchanged is tagged with a *timestamp* (time of its generation) and an expiration time *expires* (validity of the context information), after which the information is considered invalid. An example of a Context Representation Message is shown in Figure 2.6.



```
<contextML>
    <ctxEls>
        <ctxEl>
            <contextProvider id="PersonalDataProvider" v="1.0" />
            <entity id="anderson" type="username" />
            <scope>attributes</scope>
            <timestamp>2015-06-30T14:57:18-03:00</timestamp>
            <expires>2115-06-30T14:57:18-03:00</expires>
            <dataPart>
                <par n="height">179</par>
                <par n="weight">84</par>
                <par n="shoeSize">42</par>
            </dataPart>
        </ctxEl>
    </ctxEls>
</contextML>
```

Figure 2.6 – Context Element example messages (update)

2.1.5 Overview of Context Data representantion

In Figure 2.7 an overview of how context data is represented in the system is shown. Each element is tied by entity ID, entity type and scope. Each one belongs to only one provider, and has timestamp, expiration date and location information. It can store as many pairs of name and value as it needs, e.g. a location data has pairs of latitude, longitude and accuracy values.

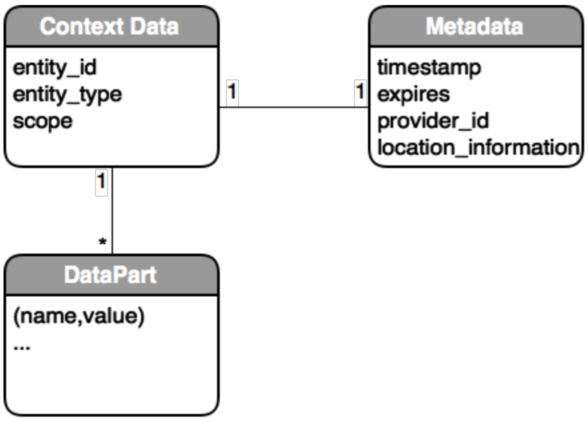


Figure 2.7 – Context Data representation

2.2 Dependability

The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable (AVIŽIENIS et al., 2004), i.e., failures will eventually happen, and the system will try to avoid them compromising the correctness of the service. Dependability is an integrating concept that encompasses the following attributes (AVIŽIENIS et al., 2004):

- Availability: readiness for correct service
- Reliability: continuity of correct service
- Safety: absence of catastrophic consequences on the user(s) and the environment
- Integrity: absence of improper system alterations
- Maintainability: ability to undergo modifications and repairs

2.2.1 Fault Tolerance

Many means can be developed to attain the various attributes of dependability and security. Those means can be grouped into four major categories (AVIŽIENIS et al., 2004):

- Fault Prevention: means to prevent the occurrence or introduction of faults
- Fault Tolerance: means to avoid service failures in the presence of faults
- Fault Removal: means to reduce the number and severity of faults
- Fault Forecasting: means to estimate the present number, the future incidence, and the likely consequences of faults

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted. This work focuses on **fault tolerance**, which is aimed at failure avoidance, and is carried out via error detection and system recovery.

2.2.2 High Availability

Looking at the dependability attributes described on the previous section, **availability** is the one this work proposes as an addition to a Context Broker system.

Any loss of service, whether planned or unplanned, is known as an **outage**. **Downtime** is the duration of an outage measured in units of time (e.g. minutes or hours) (WEYGANT, 2001).

A system is expected to be highly available when life, health and well-being, including the economic well-being of a company, depend on it. But even the most highly available services often face outages. In these cases, the expected action is that the service gets completely restored as quickly as possible, with all its capabilities ready to operate.

Availability is measured from the user's point of view. A system is available if the user can use the application he needs (PIEDAD; HAWKINS, 2008).

Availability is important for systems that require that its services are available for clients for most part of its lifetime, e.g. a web commerce system should not have an extended downtime, as money is lost on possible transactions; or a financial institution, that needs to be able to transfer funds at any time of the day, seven days a week. Some systems may also require a different approach to high availability: a window of service, for example a system that needs to be up for the entire daylight-hours, and can go under maintenance at night, reserving this time to some recovery for example, in case of an outage.

One example of use of a highly available Broker is one that receives temperature values from sensors distributed on a building, concerning fire prevention. It is important that the Broker system is always available, as the information it stores is crucial. Another example is a Broker that stores position information from sensors on a mountain that presents mudslide danger, and handles this data with the purpose of preventing this catastrophe.

High availability solutions are based on system component redundancy. If a component fails, the system is able to continue to operate using a redundant component (ENGELMANN; SCOTT, 2005). When looking for a highly available system, many solutions exist, involving replication and redundancy (ORACLE, 2015b), clustering (ORACLE, 2015a), etc.

However, when designing a highly available system, some problems may arise. Single point of failure (ENGELMANN; SCOTT, 2005), membership problem (CRISTIAN, 1991), split-brain (BARRERA et al., 1998), agreement on distributed transactions (GUERRAOUI, 2002), among others, are well-known and documented problems. Whoever is responsible for the design of the system must take action on avoiding these, to reach an optimal solution.

2.2.3 High Availability on Clusters

This work uses ideas from highly available Cluster systems to structure the highly available Broker system. For that, basic definitions needed to provide the complete understanding of the system are presented.

2.2.3.1 Cluster goals

A cluster is a collection of computer nodes that work together to provide a much more powerful system. To be effective, the cluster must be as easy to program and manage as a single large computer. Clusters have the advantage that they can grow much larger than the largest single node, they can tolerate node failures and continue to offer service, and they can be built from inexpensive components (BARRERA et al., 1998).

Some general goals of a Cluster (BARRERA et al., 1998):

Commodity: a cluster runs on a collection of off-the-shelf computer nodes interconnected by a generic network

Scaling capability: adding applications, nodes, peripherals, and network interconnects is possible without interrupting the availability of the services at the cluster

Transparency: a cluster presents itself as a single system to clients outside the cluster. Client applications interact with the cluster as if it were a single high-performance, highly reliable server. The clients as such, are not affected by interaction with the cluster and do not need modification

Failure control: ability to detect failures of the hardware and software resources it manages

We can use abstractions of nodes and resources in clusters, as nodes communicate via messages over network interconnects, and use communication timeouts to detect node failures; and a resource represents certain functionality offered at a node (BARRERA et al., 1998).

In short, a Highly Available Cluster consists of multiple machines interconnected by a common bus (AZAGURY et al., 1994).

2.2.3.2 Nonblocking protocols

Protocols that allow operational sites to continue transaction processing even though site failures have occurred are called nonblocking (SKEEN, 1981).

Crash recovery algorithms are based on the notion that certain basic operations on the data are logically indivisible. These operations can be seen as *atomic actions*.

The processing of a single atomic action is viewed as follows. At some time during its execution, a commit point is reached-where the site decides to *commit* or to *abort* the atomic action. A commit is an unconditional guarantee to execute the atomic action to completion, even in the event of multiple failures. Similarly, an abort is an unconditional guarantee to ”back out” the atomic action so that none of its results persist. If a failure occurs before the commit point is reached, then immediately the site will abort the atomic action (SKEEN, 1981).

One can find a wide variety of nonblocking protocols, e.g. two-phase (prepare and commit), three-phase (prepare, pre-commit, commit) protocols (SKEEN, 1981). This work uses as inspiration a three-phase commit protocol variation shown in (GUERRAOUI, 2002).

3 DESIGN AND IMPLEMENTATION

In this chapter the developed regular Broker solution is presented. The tools used in the development phase, the interfaces by which the system interacts with the clients, and an UML representation of the Use Cases, for a better understanding of the system workflow, are presented.

3.1 Design and Implementation of the Context Broker

This work first implements a regular Context Broker, with no fault tolerance resources. Then, it proposes a strategy to give the Broker high availability function.

3.1.1 Platform Choice

The programming language chosen for the development of this work was Python.

The system was implemented over a stateless HTTP REST (Representational State Transfer) Interface. (JAKL, 2005). For the RESTful implementation, Python Flask framework was used (FLASK, 2015). For the created web interfaces, Bootstrap was used (TWITTER, 2015) .

For data persistence, MongoDB was used.

3.1.1.1 Python, PyCharm and GitHub

Python is a powerful and easy to learn modern programming language (PYTHON, 2015). It was chosen because it represents a challenge, and to show that the system is independent of the programmed language, i.e., different applications can interact with each other in the architecture, regardless the programming language they were developed on; what matters is the content of the messages exchanged. The **PyCharm** Python IDE was used as the development environment (JETBRAINS, 2015), along with **GitHub** for version control (GITHUB, 2015).

3.1.1.2 Flask

Flask is a web application framework for Python. It is very light and easy to use. It provides RESTful request dispatching, as it is used in this work (FLASK, 2015).

3.1.1.3 MongoDB

MongoDB is a document-oriented database, classified as NoSQL. It uses a key-document data storage model, where a document can be a complex data structure. Documents can contain many different key-value pairs, or even nested documents. MongoDB is a free and open-source software (MONGODB, 2015).

3.1.2 System Architecture

In Figure 3.1 an overall diagram of the system is illustrated. Each node is a component of the system, and the arrows represent the interactions between them.

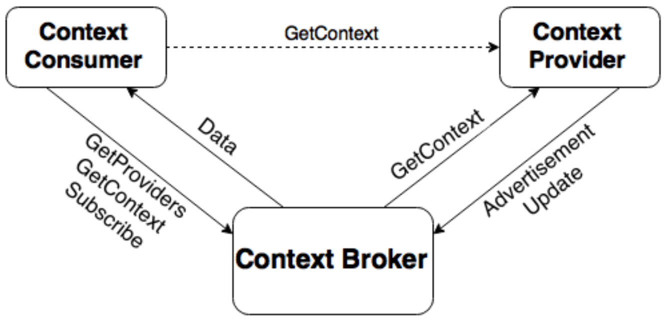


Figure 3.1 – Architecture diagram

3.1.3 Data Collections

As this work uses MongoDB to store data, the representation of Collections is used. Each Collection stores data in pair-value structure. As seen on Figure 3.2, the associations are made storing an entire object from a Collection inside another; this is made so it is easier to collect details about an associated element from another one.

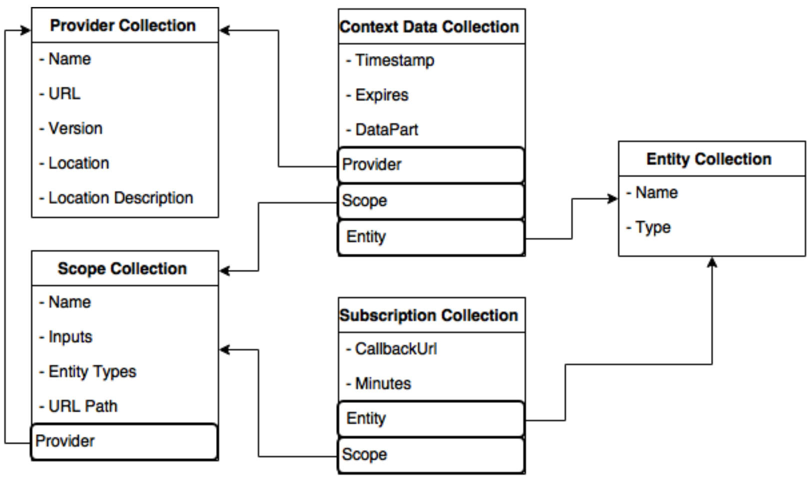


Figure 3.2 – Collections diagram

3.1.4 Broker Interfaces

The Context Broker implements several interfaces for communication with the other system components. This section presents each interface and the way they were implemented: what they expect as input (HTTP request from Consumer or Provider), the action they perform, and what they provide as output (response to the Consumer or Provider).

3.1.4.1 Advertisement (/advertisement)

Input: an Advertisement ContextML message, with Provider information, sent from the Provider

Action: registers the Provider within the Broker

Output: responds the Provider with a ACK or NACK ContextML message, informing success or error, with the corresponding error message

3.1.4.2 Update (/update)

Input: a ctxEl ContextML Message, with context information to be registered in the Broker, sent from the Provider

Action: registers in the Registry Table the context information, with its *contextProvider*, *scope* and *entity* information, *timestamp* and expiration date (*expires*) of the information. It also checks if a **Subscription** exists for the updated information, sending it to the Consumer *callbackUrl*, when applied

Output: responds the Provider with a ACK or NACK ContextML message, informing success or error, with the corresponding error message

3.1.4.3 Get Providers (/getProviders)

Input: *scope* (mandatory) and *entity type* (optional) arguments in the URL, sent from the Consumer

Action: looks for registered Providers that provide information matching the arguments given

Output: responds the Consumer with Providers Lookup ContextML message, containing a list of the providers that match the requested arguments

3.1.4.4 Get Context (/getContext)

Input: arguments *scope* and *entity* in the URL, sent from the Consumer

Action: looks for the most up-to-date Context information in the registry that matches the entity and scope received

Output: responds the Consumer with a ctxEls ContextML message, containing the , or with a NACK ContextML message, informing the error

* As seen in Figure 3.1, there can also exist a direct *GetContext* request from the Consumer to the Provider, thus not involving the Broker. This can be done by the Consumer asking the Broker for a providers list regarding a certain *scope*, and then asking a Provider directly for the desired context information.

3.1.4.5 Subscribe (/subscribe)

Input: arguments as follows *callbackUrl*, with the URL to where the Broker sends the content it is subscribed to; *scope* and *entity*, with corresponding information the consumer wants to subscribe to; and *minutes*, with the amount of time, in minutes, that the subscription is valid, sent from the Consumer

Action: registers the subscription

Output: responds the Provider with a ACK or NACK ContextML message, informing success or error, with the corresponding error message

* It is the Broker's responsibility to control the lifetime of a Subscription, based on the *minutes* argument received.

3.2 UML representation

The interactions between the clients (Consumers and Providers) and the server (Broker) will be presented using the Unified Modeling Language, UML (OMG, 2015).

3.2.1 Use Case Requirements

To present the Use Cases, a list of requirements is provided below. These requirements are adapted from a previous work (CRIPPA, 2010), to the functions of this work.

1. Register of Context Providers
   - (a) Receive Advertisement message
   - (b) Register CxP on Providers Table
2. Context Providers Lookup
   - (a) Receive Providers Lookup request (GetProviders)
   - (b) Answer with requested Providers data
3. Subscribe Context Consumer to data
   - (a) Receive Subscription request
   - (b) Register Subscription on Subscriptions table
4. Context data interactions
   - (a) Receive Context data from a Context Provider (Update)
   - (b) Send Context data to subscribed Consumers
   - (c) Receive Context data request from a Context Consumer (GetContext) and respond

Both **Lookup** and **Register of Context Providers** services make the Broker aware of the existence of Context Providers in the network.

The rest of the requirements deal with context data provision and querying, by Providers and Consumers. The Providers see the Broker as the component where they send their context information, so Consumers can find and interpret it. The Consumers see the Broker as the centralized point from where they can get up-to-date context data.

3.2.2 Use Cases

When a request from outside the system is received, the behavior is described in a **Use Case** (BITTNER, 2002). In this work, the actors are the Context Broker, the Context Provider and the Context Consumer. The following list presents the use cases in the system. For the sake of brevity, the word Context will be omitted when referring to the actors.

3.2.2.1 Registration of Providers

Name: Register Provider

Actor(s): Provider, Broker

Objective: Register Provider from an Advertisement ContextML message received from it

Description: Validates the ContextML message against the ContextML schema, then registers the Provider and its capabilities (e.g. scopes and entity types it covers) in the Broker

Type: Primary and Essential

References: Requirements 1.a, 1.b

Sequence of Events:

1. Provider sends a POST HTTP message containing an Advertisement ContextML message to the */advertisement* interface of the Broker
2. The Broker receives and validates the message
   - If not valid, the Broker sends a NACK ContextML message to the Provider
   - If valid, the Broker registers the Provider if new, or updates its information if already existent. If there is an error during the process, a NACK ContextML message is sent to the Provider.
3. The Broker sends an ACK ContextML message to the Provider, the registration was successful

3.2.2.2 Provider Lookup service

Name: Receive Provider Lookup request

Actor(s): Consumer, Broker

Objective: Receive, validate and find Providers that match the arguments received from the Consumer, then respond to the Consumer a list of Context Providers that match the information requested (*scope*and *entity type*)

Description: The Broker is the only component of the system that has information about all the Providers, thus if a Consumer wants to know where to find a specific information (matching a particular scope or entity), it must ask the Broker for a list of Providers that provide this information. The Broker creates a Providers Lookup ContextML message with the information of the Providers that match the search criteria, and sends it to the Consumer

Type: Primary and Essential

References: Requirement 2.a

Sequence of Events:

1. Consumer sends a GET HTTP message to the Broker's */getProviders* interface, with *scope* and *entity type* arguments in the URL.
2. The Broker validates the arguments: the scope argument is mandatory, while the entity is optional
   - If the scope argument is blank, the Brokers responds to the Consumer with a NACK ContextML message, informing the *Bad Parameter* error, in the error message
   - If the scope is valid, the Broker searches at its internal information for the Providers that match the requested scope and entity type
3. The Broker creates the Providers Lookup ContextML message with the desired Providers. If no Providers match the search criteria, a NACK ContextML message is created, informing *No results found* in the error message
4. The resulting message is sent to the Consumer

3.2.2.3 Subscribe Consumer to data

Name: Register Subscription from Consumer

Actor(s): Consumer, Broker

Objective: Register a Subscription made by a Consumer, to a certain *entity id*, *entity type* and *scope* combination

Description: The Subscription system provided by the Broker is a way of a Consumer to receive any new context information as soon as it is received by the Broker, within a given entity and scope combination. The Subscription is valid for a certain amount of time, defined by the Consumer. The Consumer also informs the Broker a callback URL, to where the Broker sends the new context information the Consumer is subscribed to.

Type: Primary and Essential

References: Requirements 3.a, 3.b

Sequence of Events:

1. The Consumer sends a POST HTTP message to the */subscribe* interface of the Broker, with *entity*, *scopeList*, *callbackUrl* and *minutes* arguments in the URL
2. The arguments are validated,
   - If any is blank or the minutes value is less than one (1), a NACK ContextML message is sent to the Consumer
   - If all arguments are valid, the Broker checks if the information given in the arguments exists within the Broker. If not, a NACK ContextML message is sent to the Consumer
3. The Subscription is registered in the Subscriptions table in the Broker
4. A timer is started for the subscription
5. An ACK ContextML message is sent to the Consumer, the Subscription was successful

Name: Check if Subscription expired

Actor(s): Broker

Objective: Check if timer to a Subscription runs out

Description: When the Subscription is registered, it has a *minutes* argument that states for how long this Subscription is valid. When this time runs out, the Subscription is removed.

Type: Primary and Essential

References: Requirement 3.b

Sequence of Events:

1. The timer related to a Subscription runs out
2. The Broker initiates the removal of the Subscription
3. The Subscription is removed from the Subscriptions table

3.2.2.4 Context data interactions

Name: Receive Update message from Provider

Actor(s): Provider, Broker

Objective: Receive and store context data sent from a Provider

Description: New context data is sent from the Provider to the Broker. The Broker must store it, and check if there's any Subscription related to the data stored, sending the data if a Subscription exists.

Type: Primary and Essential

References: Requirement 4.a

Sequence of Events:

1. Provider sends a POST HTTP message to the */update* interface of the Broker, containing a Context Element ContextML message with the context data information
2. The Broker validates the message agains the ContextML schema
   - If the message fails the validation, a NACK ContextML message is sent to the Provider
   - If it validates, the Broker checks if the Provider is registered and if the scope receiving data is valid.
     - If the Provider is not registered or the scope is invalid, the Broker sends a NACK ContextML message to the Provider
     - If the information is valid, the Broker sees if the entity id and entity type already exist; if not, they are created
3. The data is registered in the Broker, with its timestamp and expiration time. If there had already information about this entity and scope, that is considered deprecated and this is the newest data
4. The Send context data to subscribed Consumer use case is initiated

Name: Send context data to subscribed Consumer

Actor(s): Broker, Consumer

Objective: Check if any Subscription is related to the just updated context data, and send this data to a Consumer that is subscribed

Description: After registering the new context data, the Broker looks at the Subscriptions table for a Subscription related to the entity id, entity type and scope of the new data. If it exists, the Broker sends the same Context Element ContextML message to the Consumer in the Subscription.

Type: Primary and Essential

References: Requirement 4.b

Sequence of Events:

1. The Broker checks the Subscriptions table, trying to match the entity id, entity type and scope of the context data just registered
2. If no Subscription is found, the Broker sends an ACK ContextML message to the Provider after the update
3. If a Subscription is found, the Broker sends the same Context Element ContextML message it received from the Provider to the subscribed Consumer
4. Then the Broker sends an ACK ContextML message to the Provider after the update

Name: Receive context request from Consumer

Actor(s): Consumer, Broker

Objective: Receive a request for context data

Description: A Consumer can ask specific context data to the Broker, defining a list of scopes and an entity it wants the last information about

Type: Primary and Essential

References: Requirement 4.c

Sequence of Events:

1. The Consumer sends a GET HTTP message to the */getContext* interface of the Broker, with *scopeList* and *entity* arguments in the URL
2. The Broker validates the arguments, checking if the entity and scopes requested exist in the registered data
3. If the arguments are valid, the Broker sends an ACK ContextML message to the Consumer
4. If not, the Broker makes an extra effort, as the Request Provider for context data not found in the Broker use case begins

Name: Request Provider for context data not found in the Broker

Actor(s): Provider, Broker

Objective: Having not found requested data in the Broker, it asks the Provider for this data

Description: The Broker finds the Provider responsible for the information requested, based on the *scopeList* and *entity* arguments, and sends it a request for the information

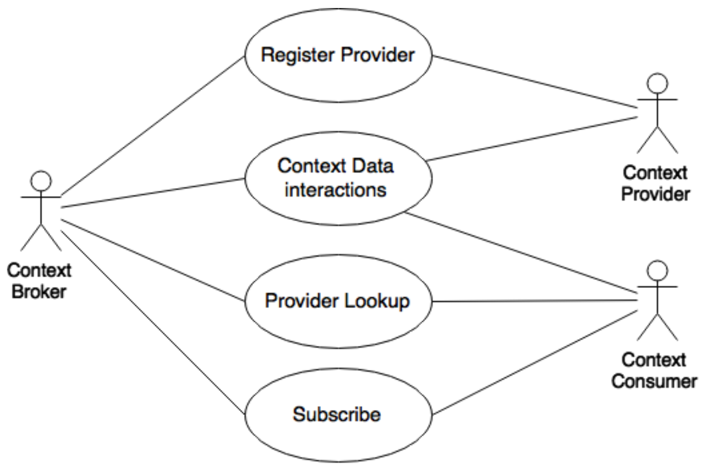


Figure 3.3 – Overview of the Use Cases

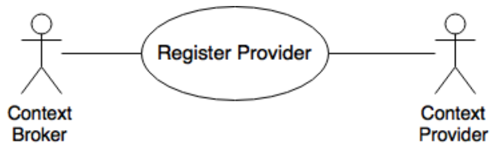


Figure 3.4 – Register Provider Use Case detail

Type: Primary and Essential

References: Requirement 4.c

Sequence of Events:

1. The Broker sends to the Provider the same context data request it received from the Consumer
2. If the Provider responds with context data, the Broker sends it to the Consumer
3. If the Provider responds with no data found, the Broker sends a NACK ContextML message to the Consumer, informing that no data was found

3.2.3 Use Cases Diagrams

A Use Case Diagram presents a good view of the actors and actions of the system. For the sake of clarity, in Figure 3.3 an overview of the use cases is presented. More detailed diagrams of the modules can be seen on Figure 3.4, Figure 3.5, Figure 3.6 and Figure 3.7

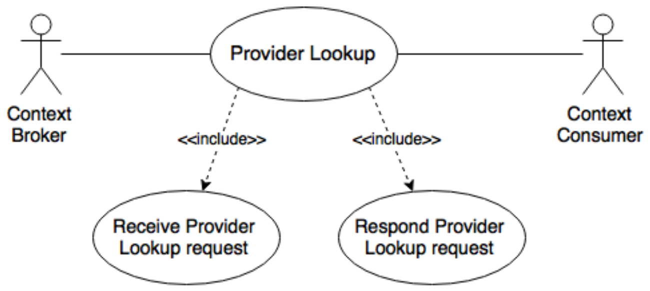


Figure 3.5 – Provider Lookup Use Case detail

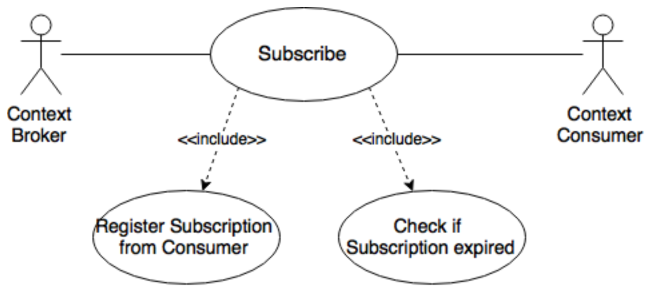

Subscribe
Context Broker
Context Consumer
<<include>>
<<include>>
Register Subscription from Consumer
Check if Subscription expired

Figure 3.6 – Subscribe Use Case detail

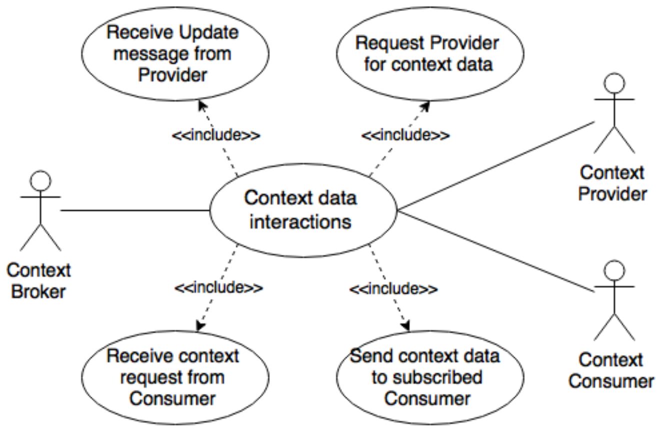

Receive Update message from Provider
Request Provider for context data
<<include>>
<<include>>
Context Provider
Context Broker
Context data interactions
<<include>>
<<include>>
Context Consumer
Receive context request from Consumer
Send context data to subscribed Consumer

Figure 3.7 – Context data interactions Use Case detail

3.2.4 Sequence Diagram

In Figure 3.8, a diagram illustrating an example of the system workflow is presented. In this execution, a Provider first registers itself on the Broker, then a Consumer subscribes for the Provider's context data, and when the Provider sends new data to the Broker (via an update), the Broker forwards this data to the subscribed Consumer. The Consumer also queries information to the Broker that it doesn't have, so the Broker asks the Provider for the information and then answers the Consumer with the data the Provider sent to it.

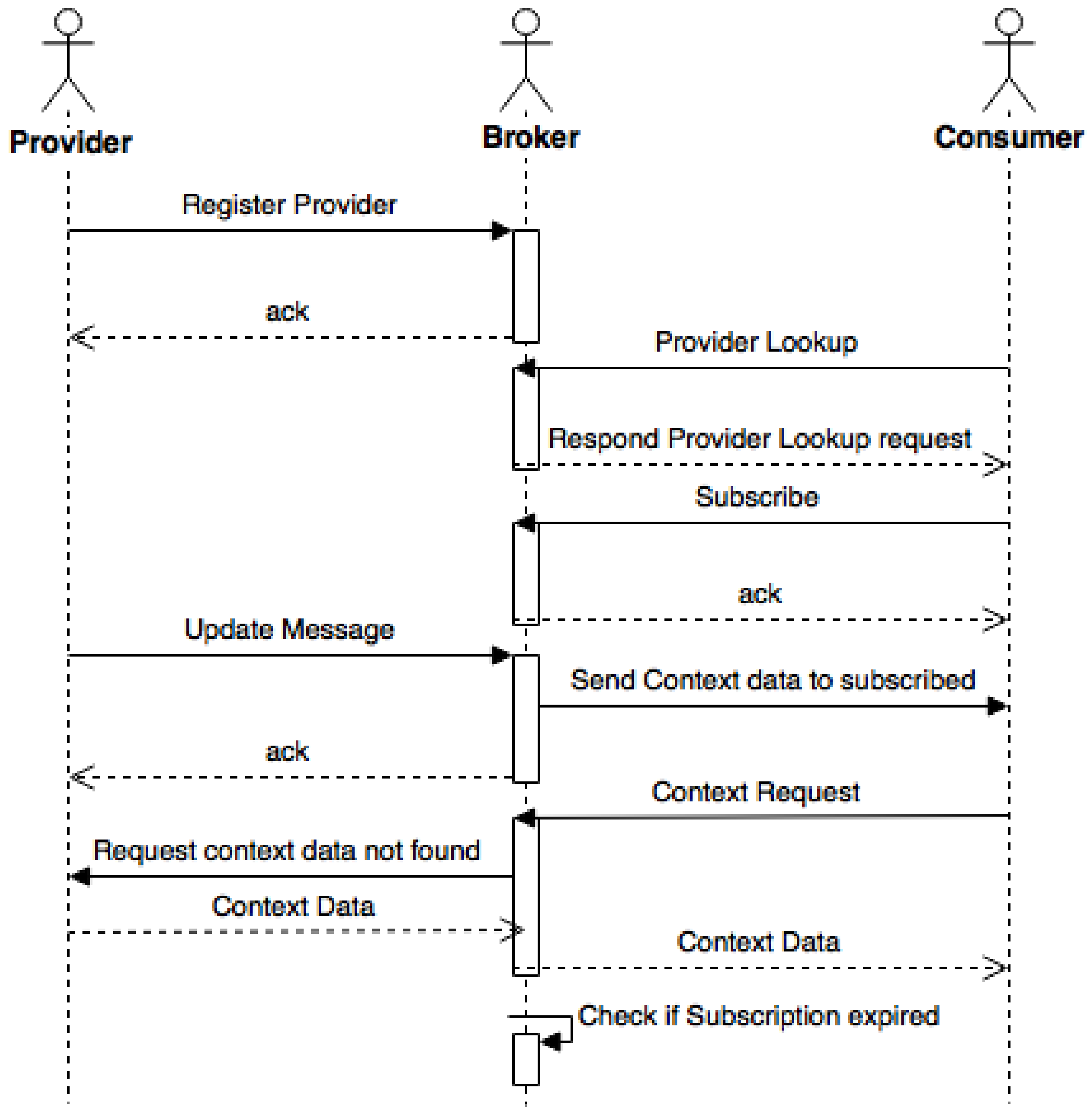


Figure 3.8 – Sequence diagram

3.3 Proposition of High Availability Technique

The objective here is to design a system of Brokers that present high availability behavior. The idea proposed here is inspired by concepts explained on Section 2.2, aiming at a cluster-like behavior from the Broker system. For the sake of brevity, both Provider and Consumer will be referred to as *client*.

3.3.1 Requirements

As availability is measured from the user's point of view, some basic requirements for a highly available Context Broker are described below:

- Clients should always be able to query data from the Broker
- Clients should always be able to insert data to the Broker
- Context data should always be consistent, i.e., different interfaces should provide the exact same information
- Every request made by a client should be answered, either by the same Broker it sent the request or by another one in the case of a failure of the first

3.3.2 Design

Following the requirements on the previous section, the idea is to use a Symetric Active/Active redundancy technique, having multiple redundant active components, with the data collection state actively replicated among them, using commit protocols. Data collection state is shared in form of a global state. This provides continuous availability without any interruption and without wasting resources (ENGELMANN; SCOTT, 2005).

The Broker System, illustrated at Figure 3.9, is composed of a predefined number of Brokers, each one with its own IP address, running independently from the others, connected on a local network. They can be accessed at the same time from different clients. The clients should have a list of the IPs from the Broker system. Each Broker may have an interface to which a client can query the addresses of the other Brokers in the same system, but the clients must know at least one of the Brokers, there is no discovery method. If a Broker fails during a request, a client won't be able to reach it, and then will try the next Broker address.

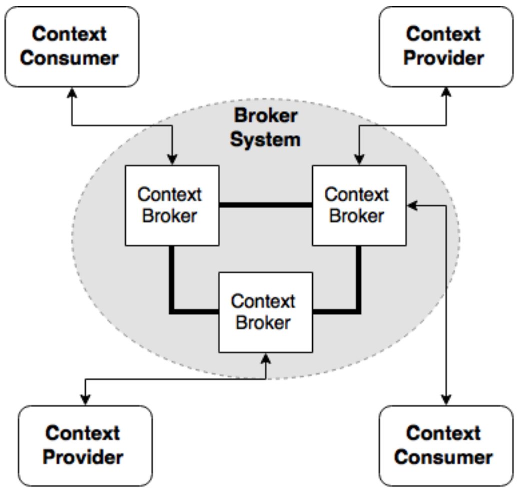


Figure 3.9 – Broker System for High Availability

All the Brokers present the same data collection, with strong consistency (VOGELS, 2009).

3.3.3 Proposed Protocol

There are two types of operations a request from a client can inflict in the data collection of the Broker system: **select** and **insert**.

When the operation is a **select**, there's no need to worry about the consistency of the data collection, as no changes are made to it. However, on an **insert**, the Broker system must guarantee the consistency of the global data collection state. This is done through a three-phase commit protocol inspired by (GUERRAOUI, 2002).

In the first phase (**prepare**), the Broker that received the requests sends a broadcast message with the request, and waits until everyone has sent an ACK message to another. The second phase (**pre-commit**) is where all the Brokers communicate to each other that they are ready to commit, i.e., ready to insert the new value in the data collection. All the Brokers wait for the ACK messages from every one. The third phase (**commit**) is where all the Brokers make the insertion in the data collection, and finally after the Broker that received the request responds its client, it broadcasts a message informing everyone that the request was completed. If a single ACK message is not received in any part of the cycle, it means one of the Brokers is down. If it is the Broker that received the request, one of the remaining ones should respond the client. The decision of which Broker responds the client falls under an election problem, a simple solution would be giving a order of priority for replacement scenarios, but the study of a more complex solution is suggested as future work. This ensures that the client is answered even if the Broker to whom the request was made goes down. The operation is not aborted, unless all the Brokers are down.

For the **select** operations, a similar idea is used, differing only in the case that there are only two phases: **prepare** and **select**. The **prepare** phase is the same as with the insert operations, all the Brokers communicate that they all have received a copy of the request; when they all confirm that received it using ACK messages, the second phase (**select**) begins. All the Brokers make the select operation on its own data collection. The Broker that received the request then responds the client and broadcasts to all the others a message informing that the request was a success, while these others don't respond the client, but wait for the confirmation that the first one did it. If any of these confirmation messages fail, the remaining Brokers will notice which Broker failed and, if it was the one that received the original request, one of the remaining Brokers responds the client, using the data it had already selected from the data collection.

In Figure 3.10 the behavior of a Broker given a particular request is illustrated using a state diagram. The following notation is used: ro_i stands for the request-original's initial state, i.e. the Broker that received the request, and b_i represents the initial state of the remaining Brokers. They may have different initial states, but they act the same way from the second state on. p, pc and c stand for the **prepare**, **pre-commit** and **commit** phases, respectively. The *commit* action can be interpreted as either **insert** or **select** atomic operation on each Broker's data collection.

The time-out value of the messages between the Brokers is an important decision: a large value is better for avoiding false-suspicions, while a small one is better for a quicker response to failures; one can use even a dynamic time-out value, that grows until it reaches a decision (GUERRAOUI, 2002). It is up to the system designer to define an optimal time-out value.

This proposed protocols is to be used for single-failure scenarios. A solution for a bigger amount of failures depends on solving the election problem when one Broker fails. Also, given the failure of a Broker, it must pass through a recovering state when restarted. A recovering mechanism for the highly available Broker system is suggested as future work.

A common problem in this kind of approach is the insertion order when two clients try to insert values regarding the same data at the same time, not leaving enough time for the message exchanges to complete. However, it is not possible in this system due to the way Context is structured. The order of insertion is defined by the client. It is not possible that two clients try to insert values regarding the same Context data at the same time, as each Context Provider

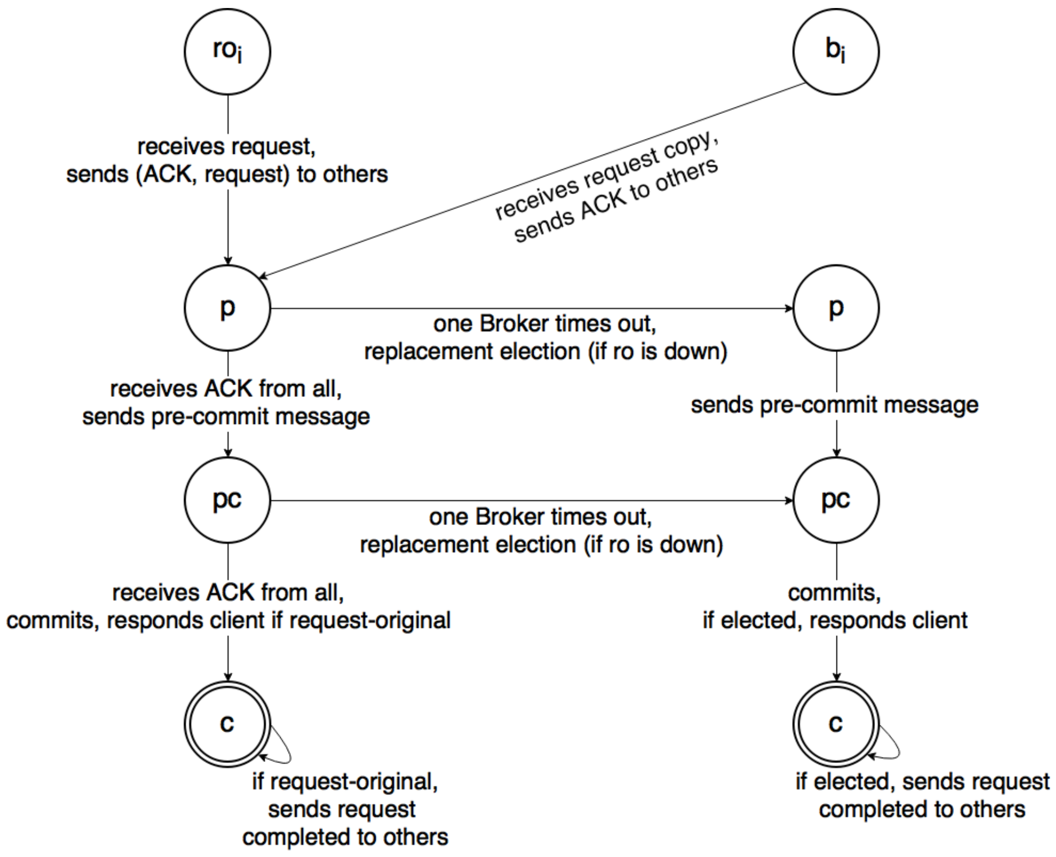

ro_i
b_i
receives request, sends (ACK, request) to others
receives request copy, sends ACK to others
p
one Broker times out, replacement election (if ro is down)
p
receives ACK from all, sends pre-commit message
sends pre-commit message
pc
one Broker times out, replacement election (if ro is down)
pc
receives ACK from all, commits, responds client if request-original
commits, if elected, responds client
c
c
if request-original, sends request completed to others
if elected, sends request completed to others

Figure 3.10 – State Diagram for proposed protocol

provides its own data, which does not interfere with other Providers data. The same applies to Consumers.

4 PROTOTYPE EVALUATION

In this work, the Context Broker was implemented. A Broker management interface was created, it is shown in Figure 4.1. For testing this implementation, both a Provider and a Consumer interfaces were created, respectively illustrated on Figure 4.2 and Figure 4.3.

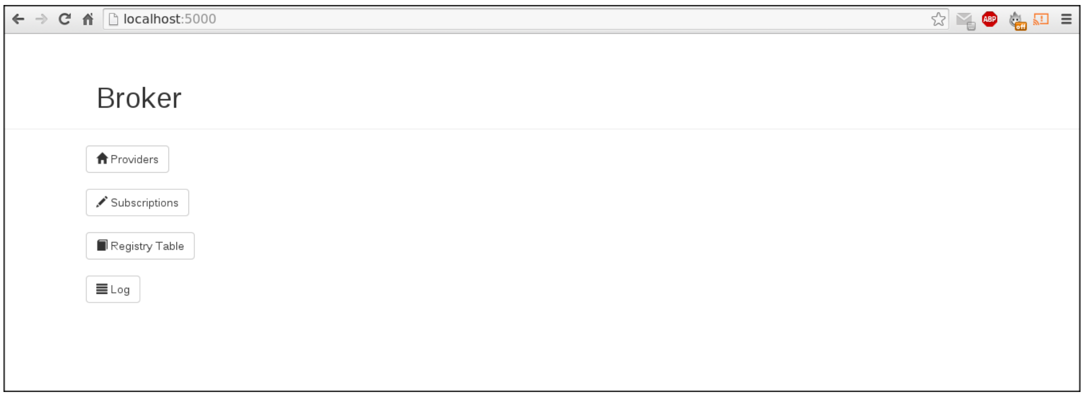
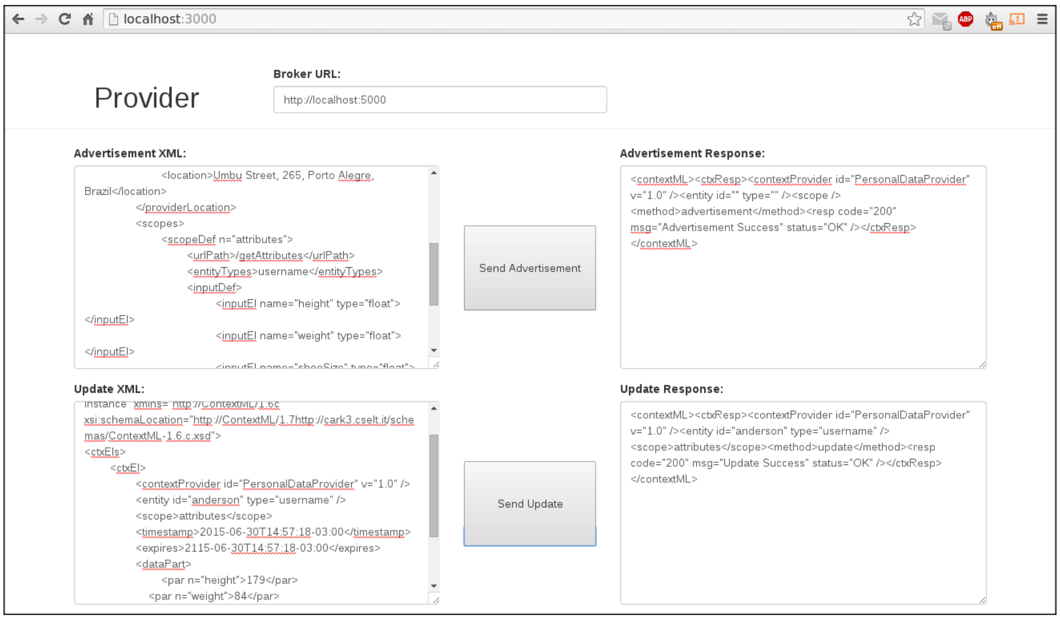


Figure 4.1 – Broker Interface



Advertisement XML:

```
                <location>Umbu Street, 265, Porto Alegre,
Brazil</location>
            </providerLocation>
            <scopes>
                <scopeDef n="attributes">
                    <urlPath>/getAttributes</urlPath>
                    <entityTypes>username</entityTypes>
                    <inputDef>
                        <inputEl name="height" type="float">
</inputEl>
                        <inputEl name="weight" type="float">
</inputEl>
```

Update XML:

```
instance" xmlns="http://ContextML/1.6c
xsi:schemaLocation="http://ContextML/1.7http://cark3.cselt.it/sche
mas/ContextML-1.6.c.xsd">
<ctxEls>
    <ctxEl>
        <contextProvider id="PersonalDataProvider" v="1.0" />
        <entity id="anderson" type="username" />
        <scope>attributes</scope>
        <timestamp>2015-06-30T14:57:18-03:00</timestamp>
        <expires>2115-06-30T14:57:18-03:00</expires>
        <dataPart>
            <par n="height">179</par>
          <par n="weight">84</par>
```

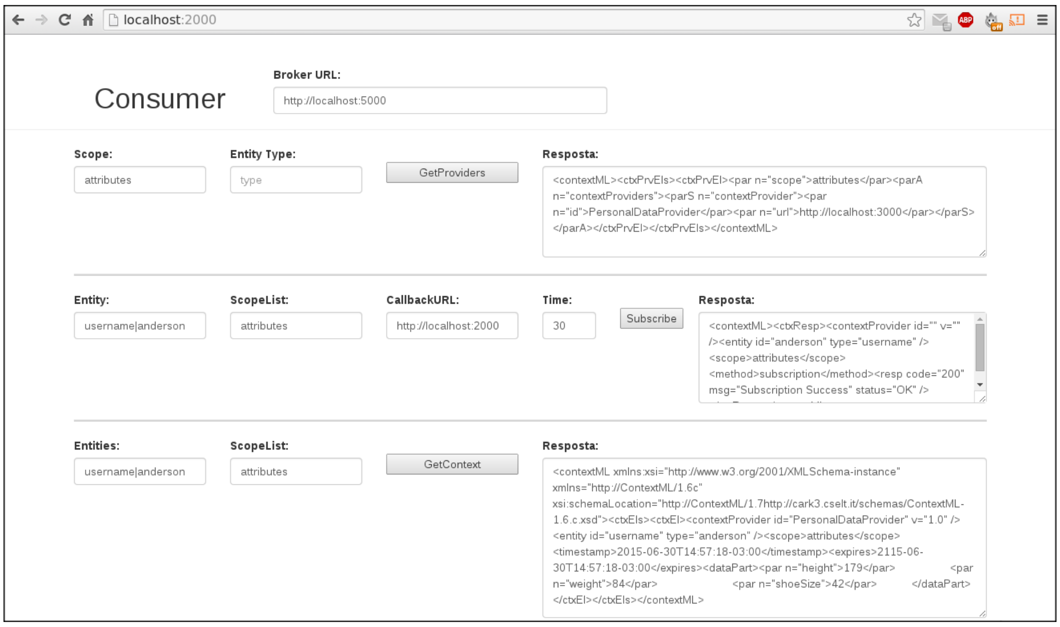
Advertisement Response:

```
<contextML><ctxResp><contextProvider id="PersonalDataProvider"
v="1.0" /><entity id="" type="" /><scope />
<method>advertisement</method><resp code="200"
msg="Advertisement Success" status="OK" /></ctxResp>
</contextML>
```

Update Response:

```
<contextML><ctxResp><contextProvider id="PersonalDataProvider"
v="1.0" /><entity id="anderson" type="username" />
<scope>attributes</scope><method>update</method><resp
code="200" msg="Update Success" status="OK" /></ctxResp>
</contextML>
```

Figure 4.2 – Provider Interface



Broker URL: http://localhost:5000

Scope: attributes

Entity Type: type

Resposta:

```
<contextML><ctxPrvEls><ctxPrvEl><par n="scope">attributes</par><parA
n="contextProviders"><parS n="contextProvider"><par
n="id">PersonalDataProvider</par><par n="url">http://localhost:3000</par></parS>
</parA></ctxPrvEl></ctxPrvEls></contextML>
```

Entity: username|anderson

ScopeList: attributes

CallbackURL: http://localhost:2000

Time: 30

Resposta:

```
<contextML><ctxResp><contextProvider id="" v=""
/><entity id="anderson" type="username" />
<scope>attributes</scope>
<method>subscription</method><resp code="200"
msg="Subscription Success" status="OK" />
```

Entities: username|anderson

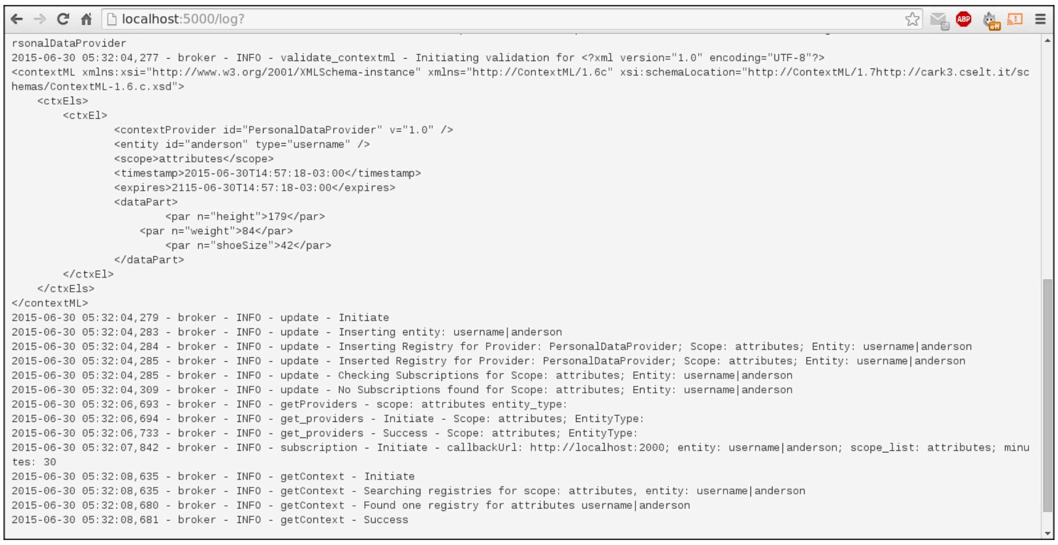
ScopeList: attributes

Resposta:

```
<contextML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://ContextML/1.6c"
xsi:schemaLocation="http://ContextML/1.7http://cark3.cselt.it/schemas/ContextML-
1.6.c.xsd"><ctxEls><ctxEl><contextProvider id="PersonalDataProvider" v="1.0" />
<entity id="username" type="anderson" /><scope>attributes</scope>
<timestamp>2015-06-30T14:57:18-03:00</timestamp><expires>2115-06-
30T14:57:18-03:00</expires><dataPart><par n="height">179</par>        <par
n="weight">84</par>           <par n="shoeSize">42</par>       </dataPart>
</ctxEl></ctxEls></contextML>
```

Figure 4.3 – Consumer Interface

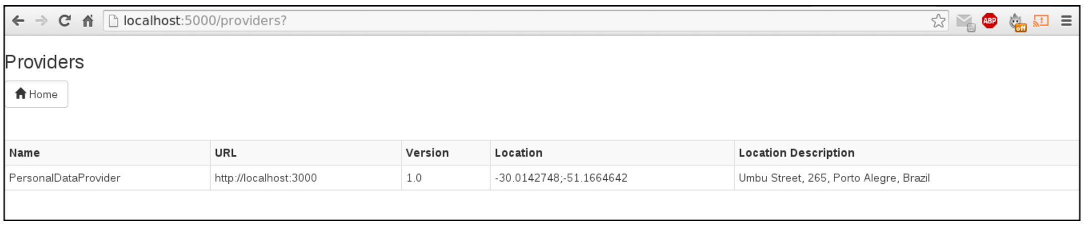
The Broker management interface allows a system manager to see the Providers registered in the Broker, shown in Figure 4.5, as well as the context information they have updated to it (Figure 4.7). The manager also has access to the Subscriptions made by Consumers and the **log** of the Broker system, shown on Figure 4.6 and Figure 4.4 respectively.



```
rsonalDataProvider
2015-06-30 05:32:04,277 - broker - INFO - validate_contextml - Initiating validation for <?xml version="1.0" encoding="UTF-8"?>
<contextML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://ContextML/1.6c" xsi:schemaLocation="http://ContextML/1.7http://cark3.cselt.it/sc
hemas/ContextML-1.6.c.xsd">
    <ctxEls>
        <ctxEl>
                <contextProvider id="PersonalDataProvider" v="1.0" />
                <entity id="anderson" type="username" />
                <scope>attributes</scope>
                <timestamp>2015-06-30T14:57:18-03:00</timestamp>
                <expires>2115-06-30T14:57:18-03:00</expires>
                <dataPart>
                        <par n="height">179</par>
                    <par n="weight">84</par>
                        <par n="shoeSize">42</par>
                </dataPart>
        </ctxEl>
    </ctxEls>
</contextML>
2015-06-30 05:32:04,279 - broker - INFO - update - Initiate
2015-06-30 05:32:04,283 - broker - INFO - update - Inserting entity: username|anderson
2015-06-30 05:32:04,284 - broker - INFO - update - Inserting Registry for Provider: PersonalDataProvider; Scope: attributes; Entity: username|anderson
2015-06-30 05:32:04,285 - broker - INFO - update - Inserted Registry for Provider: PersonalDataProvider; Scope: attributes; Entity: username|anderson
2015-06-30 05:32:04,285 - broker - INFO - update - Checking Subscriptions for Scope: attributes; Entity: username|anderson
2015-06-30 05:32:04,309 - broker - INFO - update - No Subscriptions found for Scope: attributes; Entity: username|anderson
2015-06-30 05:32:06,693 - broker - INFO - getProviders - scope: attributes entity_type:
2015-06-30 05:32:06,694 - broker - INFO - get_providers - Initiate - Scope: attributes; EntityType:
2015-06-30 05:32:06,733 - broker - INFO - get_providers - Success - Scope: attributes; EntityType:
2015-06-30 05:32:07,842 - broker - INFO - subscription - Initiate - callbackUrl: http://localhost:2000; entity: username|anderson; scope_list: attributes; minu
tes: 30
2015-06-30 05:32:08,635 - broker - INFO - getContext - Initiate
2015-06-30 05:32:08,635 - broker - INFO - getContext - Searching registries for scope: attributes, entity: username|anderson
2015-06-30 05:32:08,680 - broker - INFO - getContext - Found one registry for attributes username|anderson
2015-06-30 05:32:08,681 - broker - INFO - getContext - Success
```

Figure 4.4 – Broker Log

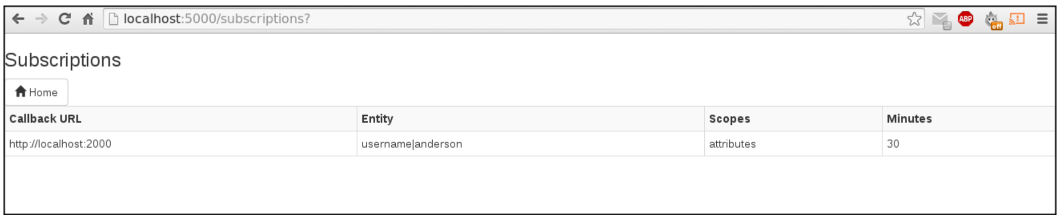
The results of single operations on each interface of the Broker, performed using the shown Provider and Consumer interfaces, are also illustrated. This was done for demonstration of the correct functioning of the Broker: a Provider was registered, made an update and a Consumer makes a subscription and queried the Broker for information the Provider had inserted. These operations created table entries on the Broker, which can be seen on Figures 4.5, 4.6 and 4.7.



Providers

Name	URL	Version	Location	Location Description
PersonalDataProvider	http://localhost:3000	1.0	-30.0142748;-51.1664642	Umbu Street, 265, Porto Alegre, Brazil

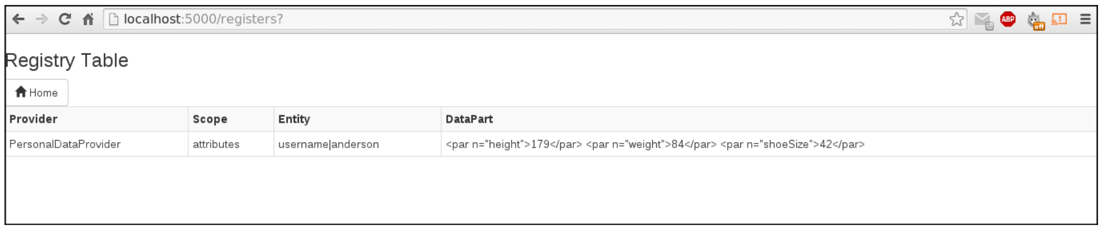
Figure 4.5 – Providers Table



Subscriptions

Callback URL	Entity	Scopes	Minutes
http://localhost:2000	username|anderson	attributes	30

Figure 4.6 – Subscriptions Table



Registry Table

Provider	Scope	Entity	DataPart
PersonalDataProvider	attributes	username|anderson	<par n="height">179</par> <par n="weight">84</par> <par n="shoeSize">42</par>

Figure 4.7 – Registry Table - Context Information

The evaluation performed was simple, regarding only the correct functionality of the Context Broker. The tests should confirm that the Broker operates correctly, without major response delays, and for a considerate amount of time without errors. The correct operation of a Broker consists on it storing and providing Context Data as it is demanded, given well-formed messages from Consumers and Providers.

For the tests, two Context Providers were created, periodically providing location context data, and two Consumers were created, querying the Broker and subscribing for data as well.

The tests results were successful. They were performed on a computer with a Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz processor, 4GB of memory and Fedora release 20 Linux operating system. The Broker was able to manage the requests and the data without major delays, with an average response time of 80ms.

5 CONCLUSION

This work introduces context and context-awareness concepts, for a better understanding of context-aware computing and how it is used. Concepts of Dependability and High Availability were also presented, to demonstrate how these are important in research and applicability. Cluster-like systems behavior was part of the inspiration to find a design for the highly available Context Broker. The implementation of the Broker was presented, as its interfaces and UML use cases description, for a better understanding of the functionality of the system. Simple tests were made, only to certify the well operating of the Context Broker. In addition of that, a protocol for integrating high availability to the Context Broker was proposed. This protocol is based on existing *nonblocking* commit protocols, and given the description of it, is not very difficult to prototype.

The goals of this work were successfully achieved. A Context Broker was developed following a previous definition, but with a different programming language (Python) and data storage mechanism (MongoDB). A different approach from (CRIPPA, 2010), but resulting in the same system from the client's (Providers and Consumers) point of view. The proposed protocol was derived from existing solutions, giving it some basis on its viability. A prototype for high availability is the natural next step.

5.1 Future Work

This work initiates a study on the development of a Dependable Broker, and naturally some future work is proposed.

Prototype for the proposed solution: development of a prototype that implements the idea presented in this work and confirms its viability, as well as looking for an optimal timeout value for the messages between Brokers

Election of Broker: study of solutions to the election of a Broker to respond the request for another Broker that has failed, using or adapting existing protocols, for a better fit in the Broker system

Recovery algorithms: study of recovery algorithms, and how to implement them on the current Broker system

Security concerns: use of HTTPS protocols, authentication method between Provider and Consumer through the Broker, types of data: public and private, requiring some kind of authentication for a Consumer to access this information

Scaling capability: make possible the addition and removal of Brokers to the Broker system, without the need to stop or restart it, dealing with membership and consensus problems

Selective High Availability: define classes of context data, regarding the need of high availability of those, making some type of data, e.g. location information with small time separation, not always highly available, what can spare some message exchanging, making the system faster

Dependability: try to attend other Dependability characteristics, e.g. safety and correctness of service

REFERENCES

ARSANJANI, A. Service-oriented modeling and architecture. **IBM developer works**, p. 1–15, 2004.

AVIŽIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **Dependable and Secure Computing, IEEE Transactions on**, IEEE, v. 1, n. 1, p. 11–33, 2004.

AZAGURY, A. et al. Highly available cluster: A case study. In: IEEE. **Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on**. [S.l.], 1994. p. 404–413.

BALDAUF, M.; DUSTDAR, S.; ROSENBERG, F. A survey on context-aware systems. **International Journal of Ad Hoc and Ubiquitous Computing**, Inderscience Publishers, v. 2, n. 4, p. 263–277, 2007.

BARRERA, J. et al. The design and architecture of the microsoft cluster service. In: IEEE. [S.l.], 1998. p. 422–431.

BITTNER, K. **Use case modeling**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

BROWN, P. J. The stick-e document: a framework for creating context-aware applications. **ELECTRONIC PUBLISHING-CHICHESTER-**, Citeseer, v. 8, p. 259–272, 1995.

BROWN, P. J.; BOVEY, J. D.; CHEN, X. Context-aware applications: from the laboratory to the marketplace. **Personal Communications, IEEE**, IEEE, v. 4, n. 5, p. 58–64, 1997.

COOPERSTOCK, J. R. et al. Evolution of a reactive environment. In: ACM PRESS/ADDISON-WESLEY PUBLISHING CO. **Proceedings of the SIGCHI conference on Human factors in computing systems**. [S.l.], 1995. p. 170–177.

CRIPPA, M. R. **Design and implementation of a broker for a service-oriented context management and distribution architecture**. 2010.

CRISTIAN, F. Reaching agreement on processor-group membrship in synchronous distributed systems. **Distributed Computing**, Springer, v. 4, n. 4, p. 175–187, 1991.

DEY, A. K. **Providing architectural support for building context-aware applications**. Tese (Doutorado) — Georgia Institute of Technology, 2000.

ELROD, S. et al. Responsive office environments. **Communications of the ACM**, ACM, v. 36, n. 7, p. 84–85, 1993.

ENGELMANN, C.; SCOTT, S. L. Concepts for high availability in scientific high-end computing. In: **Proceedings of High Availability and Performance Workshop (HAPCW)**. [S.l.: s.n.], 2005.

FICKAS, S.; KORTUEM, G.; SEGALL, Z. Software organization for dynamic and adaptable wearable systems. In: IEEE. **Wearable Computers, 1997. Digest of Papers., First International Symposium on**. [S.l.], 1997. p. 56–63.

FLASK. **Flask (A Python Microframework)**. 2015. Available at: <http://flask.pocoo.org>. Visited on July 2015.

FRANKLIN, D.; FLASCHBART, J. All gadget and no representation makes jack a dull environment. In: **Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments**. [S.l.: s.n.], 1998. p. 155–160.

GITHUB. **GitHub**. 2015. Available at: <https://github.com>. Visited on July 2015.

GUERRAOUI, R. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. **Distributed Computing**, Springer, v. 15, n. 1, p. 17–25, 2002.

HULL, R.; NEAVES, P.; BEDFORD-ROBERTS, J. Towards situated computing. In: IEEE. **Wearable Computers, 1997. Digest of Papers., First International Symposium on**. [S.l.], 1997. p. 146–153.

ICTGROUP. **Project Context Casting**. 2015. Available at: <http://www.ict-ccast.eu/>. Visited on July 2015.

JAKL, M. **Representational State Transfer**. [S.l.]: Citeseer, 2005.

JETBRAINS. **PyCharm - A Python IDE**. 2015. Available at: <https://www.jetbrains.com/pycharm>. Visited on July 2015.

KIAN, S. L. et al. A federated broker architecture for large scale context dissemination. In: IEEE. **Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on**. [S.l.], 2010. p. 2964–2969.

KNAPPMEYER, M. et al. Contextml: a light-weight context representation and context management schema. In: IEEE. **Wireless Pervasive Computing (ISWPC), 2010 5th IEEE International Symposium on**. [S.l.], 2010. p. 367–372.

MAFFEIS, S.; SCHMIDT, D. C. Constructing reliable distributed communication systems with corba. **Communications Magazine, IEEE**, IEEE, v. 35, n. 2, p. 56–60, 1997.

MOLTCHANOV, B. et al. Context management framework and context representation for mno. In: **Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2011.

MONGODB. **MongoDB**. 2015. Available at: <https://www.mongodb.org>. Visited on July 2015.

NATARAJAN, B. et al. Doors: Towards high-performance fault tolerant corba. In: IEEE. **Distributed Objects and Applications, 2000. Proceedings. DOA'00. International Symposium on**. [S.l.], 2000. p. 39–48.

OMG. **UML**. 2015. Available at: <http://www.uml.org>. Visited on July 2015.

ORACLE. **Using Clustering for High Availability**. 2015. Available at: <http://docs.oracle.com/cd/E19693-01/819-0992/fpcvr/index.html>. Visited on July 2015.

ORACLE. **Using Replication and Redundancy for High Availability**. 2015. Available at: <http://docs.oracle.com/cd/E19693-01/819-0992/gaxtb/index.html>. Visited on July 2015.

PIEDAD, F.; HAWKINS, M. W. **High Availability: Design, Techniques and Processes (Harris Kern's Enterprise Computing Institute Series)**. [S.l.]: Prentice Hall PTR, 2008.

PYTHON. **Python Programming Language**. 2015. Available at: <http://www.python.org>. Visited on July 2015.

REKIMOTO, J.; AYATSUKA, Y.; HAYASHI, K. Augment-able reality: Situated communication through physical and digital spaces. In: IEEE. **Wearable Computers, 1998. Digest of Papers. Second International Symposium on**. [S.l.], 1998. p. 68–75.

SCHILIT, B. N.; THEIMER, M. M. Disseminating active map information to mobile hosts. **Network, IEEE**, IEEE, v. 8, n. 5, p. 22–32, 1994.

SKEEN, D. Nonblocking commit protocols. In: ACM. **Proceedings of the 1981 ACM SIGMOD international conference on Management of data**. [S.l.], 1981. p. 133–142.

TUKL. **Institue for Wireless Communication and Navigation**. 2015. Available at: <http://www.eit.uni-kl.de/wicon/home/>. Visited on July 2015.

TWITTER. **Bootstrap**. 2015. Available at: <http://getbootstrap.com/>. Visited on July 2015.

VOGELS, W. Eventually consistent. **Communications of the ACM**, ACM, v. 52, n. 1, p. 40–44, 2009.

WEYGANT, P. S. **Clusters for High Availability: A Primer of HP Solutions**. [S.l.]: Prentice Hall Professional, 2001.