University of Pisa

Department of Computer Science

Master Programme in Computer Science

Master Thesis

Spray:
programming with a persistent distributed heap

Candidate

Marco Grandi

Supervisors

Prof. Vincenzo Gervasi

Prof. Antonio Cisternino

Examiner

Prof. Francesco Romani

Academic Year 2014/2015

Abstract

We introduce a programming paradigm for distributed applications based on a persistent distributed heap. A proof-of-concept implementation is provided as a JavaScript library, together with several examples that embody popular patterns for web applications.

To those I love,
especially to my grandmother Rita,
my brother Leonardo,
and Carlotta.

Acknowledgements

This work would have been much harder, and surely longer, without the contributions and help I received from many people.

I would especially like to thank my supervisors Vincenzo Gervasi and Antonio Cisternino who encouraged me to investigate persistence and distribution in programming languages. The conversations I had with them laid the groundwork of my thesis and helped me during its development.

I believe that a valuable contribution to this work comes from Vincenzo Gervasi, who gave me advice on how to write a scientific paper and suggested I adopt a shallow approach to persistence.

I thank Andrea Canciani, who gave me his previous research proposal about persistence. It was a good starting point for my study on the state-of-the-art of persistence. I am also particularly grateful for the assistance given by Simone Zenzaro, who helped me to understand the Abstract State Machine method.

I would like to thank my friends who closely shared with me the time spent working on the development of a thesis. I would especially thank Daniele Virgilio, who supported me at the beginning of this work with several hints about how to face a thesis, Marco Ponza, who proved one more time to be a priceless adventure companion, Nicola Corti, who always made coffee for me and whose laugh is contagious, and Alessandro Lenzi, who was an amusing desk neighbour.

Thanks to the "Il Parcheggione" juggling group, especially the President Cristiana, Federica, Massimo and Valentino, with whom I enjoy my free time. Thanks to the "Food Bytes" community, with whom I enjoyed tasty food and had a good time, and to "Geckosoft", namely Fabio and Davide, who supported me at the end of this work. I also greatly appreciate the support of all the other friends I have in Pisa.

My family and all my Venetian friends deserve special credits for supporting me through the entire development of this thesis.

Finally, I would like to express my gratitude to Carlotta, who has always believed in me and accompanied me to the end of this path in my life.

Contents

Introduction

> *"Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."*
>
> – Donald Knuth, *Literate Programming*

Persistence and distribution of state are two important aspects in modern distributed applications. Although they may be considered two orthogonal concepts, because persistence concerns the extent in time of state, while distribution concerns its extent in space, they are in fact closely connected.

In distributed applications, these two aspects are combined in order to let users access the application from different locations at different times. In general, these applications preserve their state after being used and restore it the next time they are accessed, without regard of locations. However, the state of an application may change from one run to another due to other users or applications that may interact with it.

A well-known example of distributed application is a webmail, which lets users read their emails from various devices through a web browser. The emails of a user are stored in servers and are retrieved by the application when the user requires them. The set of emails of a given user forms the most prominent part of the state of the webmail application for that user and changes whenever a new email, which has been sent by another user, is received.

The view that a user has of a distributed application as single logical entity, regardless of the fact that is is accessed at different times from different locations, comes at a price for the programmer who has to develop the application. The state of such application must be explicitly persisted across executions and transmitted where it is required. The details concerning these operations burden the programmer and distract him/her from the business logic of the application.

We have observed that this scenario is common to almost all distributed applications and also to other applications which are not commonly considered distributed, still transmit part of their state to remote servers in order to preserve it. We believe that programmers need an abstraction which combines persistence and distribution, simplifies the development of this kind of

applications and removes from the programmer the explicit management of aspects regarding how state is persisted and distributed.

Furthermore, we think that the programming language itself, in which the application is written, should provide the needed abstraction, making the problem interesting from a technological point of view as well as for the field of programming languages.

Distributed applications have become popular thank to the Internet and the Web, in particular. From the late 1990s, the Web has grown enormously both in amount of pages and in number of users. There have been different technological innovations that have changed the way users interact with web pages and what a web page can do. Over the years, former plugins for web browsers, such as the Java applet plugin (which were used for providing interactive features to a web page) or Flash Player (which were used to embed animation on a web page), have been partially or completely replaced by other technologies.

The idea of having "web applications", and therefore moving from static web pages to more dynamic and interactive ones, has been one of the main drivers in the development of the Web. The well-known term Ajax (that stands for asynchronous JavaScript and XML) was coined[1] to refer to a group of technologies used to create asynchronous web applications. Gmail, Google's webmail application, was one of the first popular web application developed using Ajax.

More recently, HTML5 has been introduced. It provides graphic, multimedia and interactive capabilities allowing developers to write web applications without using plugins. It is adopted for developing the web version of the applications provided by the so called apps ecosystems. An apps ecosystem is a set of related applications developed by a company, for instance Google or Microsoft. These ecosystems allow sharing information or documents among their applications and user that use them.

A main characteristic of web applications is storing the state of an application on the server side. The client side of the application, which runs on a web browser, retrieves it from the servers when it is needed. Changes applied to the state are transmitted back to the servers, where they are persisted.

Web applications are a common example of an application where the most prominent part of the state is continuosly synchronized with the server, but other kinds of application can have similar characteristics. For instance, mobile applications may store a part of their state on remote storage, both for backup purposes and for retrieving it from another device. The idea of using remote storage for generic user applications has become popular thanks to the so-called "cloud computing",

A cloud service usually provide a storage solution where applications can store their data. In some cases this storage is integrated with a number of

[1] `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/`

applications in order to persist their data. Also desktop applications may take advantage of the cloud storage and use it for storing their data. As an example, on Mac OS X applications like iPhoto, Contacts and Notes may use the iCloud storage to save their data. Besides these applications, also photos taken with an iPhone may be saved on the iCloud storage, making the service usable from both computers and phones.

The different types of applications just described emphasize how persistence and distribution are correlated in (distributed) modern applications and point out that these two aspects are, in a sense, transparent for the user. However, they must be handled by the programmer during the development of the application.

Languages used for developing these application do not support persistence and distribution as features, and so the programmer must explicitly manage them through suitable APIs and external systems. Persistence is usually achieved adopting databases, while distribution is obtained using a specific library providing primitives to communicate among the components of the application, which are connect through a network. In such a scenario the programmer has to deal with several issues, for instance how data is stored into a database and how it is transmitted.

The purpose of our work is to introduce a programming paradigm for distributed applications based on a persistent distributed heap. In our view, this is the right abstraction needed to think about persistence and distribution of state in that kind of applications and to remove from the programmer the management of the details related to these aspects.

In this thesis we present *Spray*, a programming paradigm based on a persistent distributed heap. We describe how the persistent distributed heap is organized and provide the specification of its operations. We also present a proof-of-concept implementation in JavaScript and validate our paradigm through examples written on top of our implementation. These examples embody popular patterns for web applications.

Outline

The thesis is organized as follows:

Chapter 1, Problem discusses the problem we want to solve and its feasibility, explaining how languages deal with persistence and why there is a necessity to address this feature together with distribution.

Chapter 2, Background introduces the background for both persistence and distribution, presenting approaches from scientific literature that provide these features within a programming language or as external systems.

Chapter 3, Spray programming introduces the programming paradigm based on a persistent distributed heap and present its specification in the Abstract State Machine formalism.

Chapter 4, SprayJS describes the implementation of Spray in JavaScript, motivates the choice of language and discusses some interesting details about the implementation.

Chapter 5, Examples examines popular patterns for web applications and describes their implementation in SprayJS in order to validate the *Spray* programming paradigm.

Conclusions summarizes the contributions of this thesis, presents the conclusion of the work and points out some ideas and other aspects to extend and improve the specification of *Spray*.

1

Problem

> *"As Wittgenstein observed, it is difficult to say, in advance, exactly what characteristics are essential for a concept."*
>
> – Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*

Programming languages are used to express programs, a sequence of instructions meant to perform a specific task or solve a given problem with a computer. A powerful programming language is not just a notation to write programs but "also serves as a framework within which we organize our ideas"[Abelson et al., 1996] as a result of the abstractions and features it provides.

The choices made during the design and the evolution of a language impact on its usage. Our focus is mainly on the key concepts of a language rather than on its syntax. A programming language provides a set of basic constructs and ways to combine them. On top of these, additional features can be supported and some conventions can be enforced. All these aspects affect in practice the expressivity of a language, thus coding and resource usage can be more efficient in some languages compared to other. Of course, every Turing-equivalent language has the same expressive power from a theoretical point of view. However, the ease with which computations can be expressed can vary greatly.

One of the most important resources that a programmer has to deal with is memory. All data manipulated during the execution of a program is stored in memory. How it is stored and used depend on the particular language take into account, but simplifying the management of the data in memory is an important issue in language design. A feature that most modern programming languages provide is garbage collection, a mechanism that remove from the programmer the task of tracking which objects need to be kept in memory and which ones can be destroyed.

A successful abstraction about data introduced in programming languages is the idea of "object", first appeared in Simula 67[Dahl et al., 1968].

This had led to the well-known object-oriented programming paradigm where the real world is modelled as a set of objects. The fundamental concepts of object-oriented programming are encapsulation, inheritance and dynamic methods binding. Encapsulation refer to the fact that an object contains state in form of fields (as known as attributes) and code in form of methods. Inheritance is meant to share behaviour among objects. Finally dynamic methods binding allow to change behaviour depending on the actual object. There are other characteristics about OOP, but they are not essential for our discussion.

Our problem concerns the lifetime of objects in a object-oriented programming language. A key aspect about object-oriented programming, important in our formalisation, is the fact that an object has an identity [Khoshafian and Copeland, 1986]. Identity is what allows us to distinguish two different object even if their state is equal. Such property is relevant because object's identity identifies the object during its entire lifetime.

1.1 Persistence is not a concern of programming languages

Programming languages are described as formal languages with a proper syntax and a description of the semantics. In classical textbook likes [Scott, 2009, Gabbrielli and Martini, 2010] they are explained through the basic concepts on top of which every languages are designed. These concepts are names, bindings, scoping, lifetime and memory management.

A name is a symbolic identifier[1] used to denote an entity. In the context of programming languages, an entity can be any value, location in memory, function or object. Bindings are associations name-entity. It is important to distinguish between a name and the entity to which it refers. The period of time between the creation and the destruction of an entity is called the entity's lifetime. Similarly, the period of time between the creation and the destruction of a name-to-entity binding is the binding's lifetime. These lifetime need not necessarily coincide. Also, in some programming languages, multiple names can be bound to the same entity.

The lifetime of an entity generally corresponds to the storage allocation class where it is stored. We can distinguish two main kind of allocation strategy: static and dynamic. The class of storage allocation usually languages provide are:

static allocation, used for entities which lifetime lasts for the entire execution time of a program;

dynamic allocation, used for entities created dynamically at run time. Two classes of dynamic allocation exists:

[1] In many programming languages, character sequences are used as names

stack allocation, used to allocate entities when entering in a block, procedure or function. These entities are later deallocated on exit;

heap allocation, that can be used to allocate and deallocate entities at arbitrary times.

Moreover we can distinguish memory management between manual and automatic. Static and stack allocations are managed automatically by the runtime of the language, whereas heap allocation could be either manual or automatic.

All these classifications concern management of data during the execution of an instance of a program: process. A process can be defined informally as "a running piece of code along with all the resources that the code can affect or be affected by" [Tanenbaum, 2008]. The lifetime of an entity is contained within the lifetime of the process which has instantiated it. In fact all the resources that a process has acquired are returned to the operating system when it is terminated. It follows that other systems, external to the programming language, have to be taken into account if the programmer wants to make data persistent.

Persistence is a characteristic of a system. We call persistent those systems in which entities can have a lifetime going beyond the process which has created them. We also describe as persistent those entities with such a lifetime. In classical introduction about programming languages persistence is not taken into account. Furthermore, almost all programming languages in common use today support and rely on external systems to save and retrieve data.

In other words, the persistence state of an application is managed outside the language boundary. We think that persistence should be a matter of programming languages. However, our interest about persistence is restricted to object-oriented languages. In these languages, persistence forces to guarantee object's identity across executions.

1.2 How to deal with persistence

In most programming languages, especially in object-oriented ones in common use today, to achieve persistence programmers have to use external system[2]. Here we want to briefly report what systems exist and what weaknesses they have compared to programming languages. For a more precise description we refer to Chapter 2.

[2] It is worth noting that some languages, part of which have also an historical importance such as COBOL, have been designed in order to support persistence. Conversely, the C language have been designed not considering input/output, which have been left out from the language and provided through a library. The success of this language have led to assert the idea not supporting input/output in the definition of a language.

The problem of how to deal with persistence in programming languages is not new and it has been addressed in the literature in the context of persistent programming. The most ambitious approach in this area was the one introduced under the name of *Orthogonal Persistence*. The concept of this methodology is "to identify persistence as an orthogonal property of data, independent of data type and the way in which data is manipulated" [Atkinson et al., 1983]. However, programming languages supporting this concept have been used to develop programs only in the academic environment. Other solutions to have persistent state are commonly in use today.

When the amount of data is huge, the common system used is a database management system (DBMS). Data is save on databases, which are organized following a precise logical model. These systems are the most used in commercial development because they provide a reliable way to manage data. The problem these systems raise is that the programmer has to manage the different representation between the language and the database. In the case of an object-oriented language and a relational database, the problem is known as object-relational impedance mismatch. This problem can be limited using middleware that provides an object-relational mapping (ORM), but can not completely be eliminated. From a language prospective an ORM simplifies the development of a program but does not guarantee the object's identity. An example of ORM is Java Persistence API (JPA), an interface that describes the management of relational data in Java and that is implemented in the Hibernate middleware.

Another form of persistent storage is represented by files. They are the most common low-level solution to save and retrieve persistent state. Programming languages expose APIs to deal with files and file system. A file is generally intended as a sequence of bytes[3] and the process of transforming an object in a sequential representation suitable for being saved on a file is called serialization. Serialization can be performed manually by the programmer, who has to manage properly how data is converted from the language representation to the sequential one. This process is executed through a sequence of write operations on the file. The other option is to perform serialization by means of automatic systems, generally related to the concept of reflection. Most modern programming languages, especially object-oriented ones, provide a library with a serialization algorithm that keeps the programmer away from the management of the representation.

Object-oriented programming languages generally identify an object by its address in memory. This choice combined with an external system to provide persistence does not guarantee long-term object's identity. Indeed when an object is saved outside the language boundary its address is not commonly saved together. The exact address in memory where an object is

[3]Other organizations are possible as well, for instance in COBOL there is also the indexed organizations that can be accessed directly through an index.

stored depends heavily on a set of factors, including the operating system. As a consequence what is understood as a unique object is instead represented by two different objects in two distinct executions. To fix this problem we have to include persistence in the definition of a programming language.

Orthogonal persistence has been introduced in 1983 [Atkinson et al., 1983] and was an active research topic until the late 1990s. Its main purpose was to add persistence as a programming language feature and therefore to yield programming languages able to express persistent data. The principles of this approach have not been changed across the years and the main goal of the research was to produce a programming language that provides orthogonal persistence suitable to be used outside the academia. Although this approach has not directly influenced any modern programming language, it represents a good starting point to discuss persistence.

1.3 A change of perspective

Applications can be informally described as programs designed to perform a task. As such, they are not directly related to an execution state or to the resources being used. Instead, they tend to depend on some form of persistent state. These characteristics are in contrast to the ones that a process has, leading up to a greater effort for the programmer. It should be noted that an application can have a state which makes the user perceive it as "running" even though it has no active process in the operating system.

As an example, in Android, Google's mobile operating system, the process hosting an application can be killed, if memory is needed, after that another one has taken the focus. The application can persist unsaved changes before being paused. If the hosting process was killed and the application takes back the focus, another instance is created. This instance received the saved state, if it was persisted, and can restore it. As a result, the application appears to the user in the same state as when it was paused, even if its process was terminated.

Programming languages are usually designed to only express what happens within a single process[4]. In order to simplify writing applications, a language should at least lets to express what happen to data across more executions of the same program. This is possible in the languages introduced in the literature about orthogonal persistence. These languages let to have persistence of data across different executions of the same program and, is some cases, also among programs. However, sharing of persistent data was possible only within the boundary of the persistent programming system to which the particular language belongs.

The increasing interest and spread of distributed systems and distributed applications starting from the 1990s has brought out a new scenario not con-

[4]Of course there are exceptions, for example occam or Erlang.

sidered before: distributing data among different processes in execution on different systems. Such situation was not taken into account in orthogonal persistence but should be a main concern in programming languages according to the omnipresence of such systems and applications nowadays. A distributed system can concisely be described as "a software system in which components located on networked computers communicate and coordinate their actions by passing messages" [Coulouris et al., 2011]. An application that runs in a distributed system is called distributed application.

A distributed application has more components that communicate with each other and share the state. Such state needs to be made persistent, so that changes are not lost when accessing again the same application. There are two main possible architecture for a distributed application: client-server and peer-to-peer. In a client-server architecture the persistent state is usually provided by servers and it is requested by clients on demand. In a peer-to-peer settings instead the state is split among peers, any of them can make a request to another one.

The Web is certainly one of the most important distributed systems and web applications are a classic example of application with a persistent distributed state. The structure of a web application is client-server and the client-side is represented by the web browser. The state of such applications are completely saved on the server-side, but client-side replication of some information is possible. As an example web session is implemented in browsers using cookies, a technique to store in the browser a minimum amount of data. It is worth noting that browsers provide an additional persistence feature. In fact they restore the last session[5] when launched or after a crash. This feature may be easier to implement if persistence would be provide by the language used to develop the browser.

Mobile applications are applications designed to run on smartphones, tablet computers and other mobile devices. They have seen widespread adoption in recent years and are another example where persistence and distribution could be a relevant concern. Mobile devices are characterised by limited resources and run on battery, although they provide additional features, such as location detection and gyroscope, compared to traditional computers. Limited resources availability adds constraints to the development of such applications; as an example the number of processes in execution are typically limited to prevent running out of battery.

Nevertheless the operating system of these devices tries to give the illusion of having multiple application running at the same time. When the limited resources available are insufficient to keep all background processes alive, the system select an application which is notified to terminate. This application before terminating could save its state and must release acquired

[5] in this case with session we intend the list of open sites, rather than the state of each of these

resources, then the related process are terminated. This makes the resources available to the system, which can then give them to the new application. The application to which the user is switching might be a new one or not. If the application has preserved its status, the new application process is started up and it restore the previous state. This makes it possible for the application to appear in the same state as when its process was terminated. This behaviour could evidently take advantage of persistence if it would be provided by the language, removing from the programmers the effort to preserve the application status.

A considerable number of mobile application are also the client-side of a distributed system. Examples of such application are instant messaging, note taking and archiving. Distribution is used both to share information and for backup purposes due to the limited storage available on mobile devices. This aspect of the application is completely developed by the programmer because typically languages provide only communication primitives to send and receive messages along a network.

Distribution is also a fundamental trait in "Cloud computing", where both computing resources and storage are shared by multiple users. A key characteristic in the cloud is device and location independence, that enable users to access the systems regardless of their location or what device they use. Applications are accessed via the Internet and their state has to be distributed to clients (both web browser or native app).

1.4 Goal: persistence and distribution provided as language feature

In summary, most programming languages do not provide persistence as a language feature to preserve data among execution or supply an abstraction for masking send/receive primitives in order to transparently distribute data between components of a distribute application.

On one hand there are languages that provide persistence as a feature, such as those that enforce orthogonal persistence. On the other hand there are languages that provide distribution through elementary send/receive primitives, such as Erlang, but this kind of communication is typically limited to the process boundaries[6]. However, there are not programming languages that provide persistence and distribution together.

We restrict our problem to object-oriented languages because they have a built-in concept of identity, as we have previously mentioned. This choice is not limiting since most modern programming language are object-oriented and tend to prefer supplying additional features and mechanisms to simplify the development of a program at the cost of a small loss of performance. On

[6]Communication across a network is commonly provided through a set of APIs of a library, and it is not taken into account in the language's semantics.

the other hand identity lets us handle persistence and distribution at the same time. These two concepts are indeed orthogonal, but a more extensive concept of identity allows referring to the same object among executions and from different locations.

1.5 About a solution

Persistence and distribution in our formalisation are discussed at the programming language level, but other choices are possible. The most viable alternative is to address these two aspects at the operating system level since an operating system already provides persistence through its file system, so the main concern is distribution. This approach was at the bottom of the research about distributed operating systems. Unfortunately nowadays such way to tackle the problem is not possible because of the commercial interests behind modern operating systems.

The general trend in programming language development is instead to port a language and its tools (compiler, runtime, virtual machine, etc...) on a wide range of operating systems in order to promote its diffusion. Such trend concerns not only open source projects but also property technologies; as an example Microsoft has released .NET Core, a modular development stack at the bottom of its .NET platform.

Adding persistence and distribution as a language features leads to an obvious increase in the size of its runtime, which has to be modified in order to provide these functionality. However, one might expect that programs which do not use such features do not exhibit a notable loss in performance. Moreover, there are some aspects that should be addressed in order to explain the feasibility of a solution.

How to achieve persistence within the operating system boundary is not a real issue since methodology from the orthogonal persistence approach can be adopted. In an object-oriented language, the basic idea is to provide the language with a object storage where all persistent objects are stored. Assumed that a particular language is modified in order not to use a memory address as object's identity, what mechanism should be used becomes a significant issue. In such a scenario the identity of an object must be unique among all processes that can refer to it. This problem becomes more difficult if we consider a distributed environment, where the identity must be unique among all processes located on different components.

In order to have unique identifiers, any of which represents the identity of an object, we may adopt a central registry. However, this solution implies to introduce in the system a single point of failure that can potentially block it. Our choice is to use as object's identity a randomly generated identifier. This decision could lead to cases where the same identifier it is shared between two or more objects, in other words a form of hash collision is possible.

Given that this is a technical problem, we put it into the background of our approach and move forward in the investigation of a possible solution. However, using random identifier implies that object creation is a local action and allows each components to go along with its computation independently.

A group of related applications developed using a language that provide persistence and distribution looks like a distributed system. As a consequence the conditions of the CAP Theorem [Brewer, 2000, Gilbert and Lynch, 2002] are met. Such theorem states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

- Consistency
- Availability
- Partition-tolerance

In our formalisation, consistency means that all applications that are manipulating the same object see the same state. In other words distributed objects are atomic and there must exist a total order on all operations on an object such that each operation looks as if it were completed at a single instant. Availability implies that each application can modify a distributed object and will receive a response about whether it succeeded or failed. Finally partition-tolerance stands for the fact that an application can continue its execution even if another one that use the same objects failed or messages that refer to objects it manipulated are lost.

If we wanted to provide all three guarantees, we would certainly add to the language's runtime a synchronisation mechanism in order to achieve consistency. In doing so, we are implicitly assuming that the underline network is reliable. Although there are reliable protocol for host-to-host communication, like TCP, connection errors are still possible and disconnection should take into account. As an example a device which use a mobile telecommunications technology as Internet access may have disconnections due to reception problems. As a consequence, a short disconnection of only one application causes the temporary blocking of the whole system because synchronisation among all components is not possible.

In our view the most desirable properties of such a system are that each application can apply local modification as soon as it is required and not be blocked due to other applications failure or message losses. In other words the solution has to provide availability and partition-tolerance, while the consistency requirement can be relaxed. Eventual consistency is chosen instead as consistency model so that stale state is possible in an application.

2

Background

> *"The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities."*
> – Edsger Dijkstra, *How do we tell truths that might hurt?*

How to persist and distribute data is an important issue during the development of an application or a complex system. These two aspects are in general managed separately by different mechanisms or systems. Although our presentation mainly concerns about well-known systems that are currently in use, other methods introduced in the literature are described.

In the following we will discuss both persistence and distribution. First, we present methods to persist data, either in the languages or through an external system. Since our interest is in programmer perspective, we also report examples when they are meaningful.

Later we address distribution and systems that provide it. Also in this case, we report examples of languages that treat distribution as a core feature. Thereafter we summarise the concept of distributed operating system and recall some important aspect develop in this research area.

Finally we describe eventual consistency, introduce the problem of conflicting states that can occur in distributed system and report a formal framework developed for automatic conflict resolution, namely conflict-free replicated data types.

2.1 Orthogonal Persistence

Orthogonal Persistence [Atkinson et al., 1983] is a language-independent methodology for providing persistence as a language concept. In programming language design, "orthogonality means that features can be used in any combination, the combinations all make sense, and the meaning of a given feature is consistent" [Scott, 2009]. Orthogonal Persistence applies this idea to the persistence of data, which is seen as orthogonal property, independent of data type and the way in which data is manipulated.

Atkinson and Morrison in [Atkinson et al., 1996a] recognised the following principles of Orthogonal Persistence:

- **Persistence Independence**
  The persistence of data is independent of how the program manipulates the data. That is, the programmer does not have to, indeed cannot, program to control the movement of data between long term and short term store. This is performed automatically by the system.

- **Data Type Orthogonality**
  All data object should be allowed the full range of persistence irrespective of their type. That is, there are no special cases where objects of a specific type are not allowed to be persistent.

- **Persistence Identification**
  The choice of how to identify and provide persistent object is orthogonal to the universe of discourse of the system.

It is worth reporting the footnote regarding the principle of persistence identification written by Morrison in [Dearle et al., 2009]. He stated that the experience showed that the only mechanism for implementation was persistence by reachability, also known as transitive persistence. Such mechanism consist of choosing an element as root of persistence: the root as long as all elements it transitively point out are preserved.

One of the primary motivations for Orthogonal Persistence in [Atkinson et al., 1983] was the conceptual difference between short term and long term data. Short term data is manipulated by the programming language facilities within the program execution, while long term data is required to survive to process termination. For long term data, Atkinson and Morrison direct they attention to database management system (DBMS). They highlight how data is defined using a type system within a programming language while it is described according to a data model, when it is stored in a database[1]. Furthermore, they point out that the difficulties of understanding and managing the mapping between these two representation frequently distract the programmer.

Orthogonal Persistence was introduced to provide integration between programming languages and DBMS. In a more general prospective it leads to the research in orthogonally persistent object systems where different aspects related to persistence was investigated, such as persistent stores, concurrency and transactions. A broad review about this research area is reported in [Atkinson et al., 1996a]. Moreover, Orthogonal Persistence principles have been used as criteria for evaluating systems that provide persistence. As an example [Takasaka, 2005] is a survey regarding persistence approaches for

[1] The set of conceptual and technological differences between these two models is known as impedance mismatch.

object-oriented programming languages that adopts Orthogonal Persistence principles among its criteria for the assessment.

In the next sections we report some languages that comply with the three principles introduced at the beginning of this section in order to show how persistence can be provided in a language.

2.1.1 PS-algol

PS-algol [Atkinson et al., 1982, Atkinson et al., 1983] is a persistent programming language implemented as a functional extension of S-algol[2]. It was the first language to provide orthogonal persistence, which provided persistence by reachability for all data type supported by the language. The functions added to S-algol to support persistence are: *open.database*, *close.database*, *commit* and *abort*. These procedures point out how persistence in PS-algol was understood: the integration of a database in the language.

The procedure *open.database*, as well as *close.database*, required to specify a database name, which is in general is a path down a tree of directories. The directory mechanism for databases was implemented to avoid dependence on particular operating system features. Different programs could open the same database in order to operate on the same persistent data[3]. In [Atkinson et al., 1983] some examples illustrate how small program can be composed to put up a complete system. PS-algol also provided a set of procedures to manipulate tables, that are associative stores[4]. The successful opening of a database returned a pointer to a table which is the root of persistence: preserved data is identified by transitive closure of reachability from the table.

```
structure person(string name, phone; pntr addr)
structure address(int no; string street, town)
let db = open.Database("addr.db", "write")
if db is error.record
  do { write "Can't open database"; abort }
let table = s.lookup("addr.table", db)
let p = person("al", 3250,
               address(76, "North St", "St Andrews"))
s.enter("al", table, p)
commit
```

Listing 2.1 A program that adds a new person to a database.

[2]S-algol is a programming language that stands somewhere between ALGOL W and ALGOL 68.

[3]Although many database may be open for reading, only one may be open for writing.

[4]As key was possible to use an integer or string value.

In listing 2.1 an example from [Dearle et al., 2009] is shown to give a flavour of the language. The code fragment show how to add an element to a database: the procedure *s.enter* store the person `p` in the table `table` with the key `"al"`. After that, the change made to the database are committed.

As mentioned in [Atkinson et al., 1983], "the programmer never explicitly organizes data movement but that it occurs automatically when data is used". The paper also states that "a reduction by a factor of about three in the length of the source code" was achieved compared to programs written in Pascal with explicit database calls.

2.1.2 Napier88

The Napier88 system [Morrison et al., 2000] was designed as an integrated persistent programming system. It consist of the Napier88 language and its persistent environment. As described in [Morrison et al., 2000], it "was intended as, or turned out to be, a testbed for experiments" as an example in type systems for data modelling, concurrency control and transactions and object stores[5]. As far as persistence is concerned, Napier88 provided a predefined procedure in the language to access the persistent store. Each program may access its persistent environment using that procedure. It is worth to point out that "each persistent store is organised as a graph of object but the topology of the graphs may vary from store to store" [Morrison et al., 2000].

In the listing 2.2 is shown an example from [Dearle et al., 1989]. The code fragment creates an environment called `new`, places a vector in it and finally creates a binding to the new environment in the root environment, making it persistent. The `PS` procedure returns the root of the persistent store. The names `e` and `avector` persist; they are reachable from the root of persistence.

```
let new = environment()
in new let avector = vector @ 1 of [1,2,3,4,5]
in PS() let e = new
```

Listing 2.2 A program that adds a vector in an environment and makes it persistent.

2.1.3 Persistent Java

Java [Arnold and Gosling, 1996, Gosling et al., 1996] is a well-known object-oriented programming language, in common use today. It has a strong type system and automatic memory management. Its first implementation was released in 1995 and quickly achieved considerable prominence. Researches in persistent programming was interested in Java as well due to the language's

[5]For further references look at [Morrison et al., 2000].

features and its mainstream use. Several persistence mechanism have been implemented to provide support for persistence in Java, here we report only three implementation[6] that adhere to the principles of Orthogonal Persistence but differ for how persistence is provided. Other mechanisms are discussed elsewhere: [Moss and Hosking, 1996] describe a range of approaches for adding persistence to Java whereas [Jordan, 2004] present a comparative study among different reference implementations.

One of the most prominent implementation of orthogonally persistent Java was developed in the *Forest Project* [Sun Microsystems, Inc., 2000], a collaborative research project among Sun Microsystem, where Java was originally developed, and the research group of Atkinson at the University of Glasgow. The implementation was called PJama[7] [Atkinson et al., 1996a, Atkinson et al., 1996b, Jordan, 1996, Jordan and Atkinson, 1998, Atkinson and Jordan, 1999] and provided persistence without any change to the Java language, its standard libraries or compiler. Persistence was achieved by exposing an additional API, predominantly methods of the class PJavaStore, and modifying the Java virtual machine (JVM)[8] The API allowed the programmer to associate objects with strings in a persistent map in order to make them persistent. The associated object was recorded as root of persistence and every object transitively reachable from it were preserved. The virtual machine was extended by adding an object cache and the management of the movement to and from the object store integrated in the system.

In the listings 2.3 and 2.4 two examples from [Atkinson et al., 1996a] are shown. In the former program an object is associated in the store to the string `sp1`.

```
public class SaveSpag {
  public static void main (String[] args) {
    Spaghetti sp1 = new Spaghetti(27);
    Spaghetti sp2 = new Spaghetti(5);
    sp1.add("Pesto");
    sp1.add("Pepper");
    sp2.add("Quattro Fromaggio");[9]
    try {
      PJavaStore pjs = PJavaStore.getStore();
      pjs.newPRoot("Spag1", sp1);
    } catch (PJSException e) { ... }
  }
}
```

[6] Previously reported in [Dearle et al., 2009].

[7] It was formerly known as PJava, but then that name was reserved for Personal Java, as reported in [Jordan and Atkinson, 1998].

[8] The implementation could not make all type persistent, as an example `Thread`. More details in [Atkinson et al., 1996b, Atkinson and Jordan, 1999].

[9] sic.

Listing 2.3 A program that persists an object in the store.

The latter program shows instead how to retrieve the persisted object from the store and use it.

```
public class SaveShow {
  public static void main (String[] args) {
    try {
      PJavaStore pjs = PJavaStore.getStore();
      Spaghetti sp = (Spaghetti) pjs.getPRoot("Spag1");
      sp.display();
      pjs.newPRoot("Spag1", sp1);
    } catch (PJSException e) { ... }
  }
}
```

Listing 2.4 A program that retrieves an object form the store and uses it.

An interesting fact about the history of Pjama regards the Java keyword `transient`. This keyword "suggests that the designers gave some thought to persistence" [Jordan and Atkinson, 1998]. As can been seen in the first version of the Java language [Gosling et al., 1996], the keyword was defined as reserved field modifier but no related services was provided. Only a hint was reported: "Variables may be marked `transient` to indicate that they are not part of the persistent state of an object". The lacking of a semantic meaning of the keyword allowed the researches to use it in order to specify the behaviour of a static variable,as described in [Atkinson et al., 1996b, Jordan and Atkinson, 1998]. Unfortunately, in Java 1.1 the semantics of `transient` was provided along with the support for object serialization[10] in a manner incompatible with PJama[11]. Furthermore, the project was concluded in September 2000 with the publication of [Atkinson and Jordan, 2000].

The second variant of Persistent Java that we want to discuss is the one implemented on the Grasshopper operating system [Dearle et al., 1996]. Grasshopper was an operating system that provided support for Orthogonal Persistence so no modifications to Java were needed in order to add persistence. It was sufficient to instantiate "the entire Java machine within a persistent address space" [Dearle et al., 2009]. Although this solution does not strictly comply with our request to address the problem at a language level, it is interesting because show that other approaches are possible.

The third and last implementation of Persistent Java is ANU-OPJ [Marquez et al., 2000], that stands for Australian National University Orthogonal Persistence for

[10] Serialization is the mechanism that Java provides for persistence and distribution

[11] More details can be found in [Jordan and Atkinson, 1998].

Java. The design of this approach was driven by three features: complete transparency of persistence, support for both intra and inter application concurrency and the preservation of Java portability. Complete transparency was achieved by interpreting the third principle of Orthogonal Persistence more fully to mean that the roots of persistence should also be defined implicitly. Thus class variables[12] was made implicit root of persistence, making persistence truly transparent. Portability was preserved by using a customised class loader that semantically extended programs at class loading time, making the approach compatible with standard compilers and virtual machine. The portability of the storage layer was guarantee as well, providing a clean interface between the Java runtime and the underlying store. Unfortunately, ANU-OPJ could not uniformly perform byte code transformation on some system classes, similarly to PJama.

2.1.4 Persistent Oberon

Persistent Oberon [Bläser, 2006, Bläser, 2007] is a persistent language derived from Active Oberon[13]. It is based on a modular object-oriented programming model, which combines the notion of object with the the concept of modules. It offers persistence as a naturally inbuilt concept: the "language does not need any artificial programming interfaces or commands to use persistence" [Bläser, 2007].

In Orthogonal Persistence the fundamental concern is to avoid distinguish between transient and persistent data because such distinction lead to problems: an external system is needed, the mapping between the language and the system has to be managed and sometimes also the movement of data. Although there are languages that comply with Orthogonal Persistence principles, in [Bläser, 2007] is stated that "regrettably none of them fulfils this goal of language-instituitionalized persistence". Bläser claims that special program functions and explicit interfaces are required in these language. His interpretation of persistence identification is similar to the one adopted in ANU-OPJ. Furthermore, "Persistent roots have to be handled entirely differently in comparison to the transient ones". These statement can be confirmed by the listing previously reported.

Persistent Oberon have been designed in order to overcame these problems, but in a manner different from the one chosen in ANU-OPJ. The module, besides being a static compilation and deployment unit, is the root of persistence and it is designed to live infinitely long in the system. All transitively reachable object in a module are preserved.

[12] Fields of a class declared with `static` modifier.

[13] Active Oberon is the object-oriented descendant of Oberon.

2.2 Files and file system

Files are an abstraction mechanism provided by the file system, which is one part of the operating system, that allow storing information on the underlying storage devices. From a programmer perspective, files are the most common low-level form of persistent storage, as they are resources which are preserved across process and system restart.

A file is an unstructured sequence of bytes and does not enforce any particular organisation of data, but it only provide very simple operations:

- **sequential operations**: *read/write* a block of bytes from the current position of *seek* to a position in the byte sequence;
- **map operations**: *map* a contiguous block of bytes from the file in memory; all of the access to the memory range conceptually happen on the file.

In general, programming languages provide a set of API to have access to the underlying file system and manage files. To be more specific, they allow to travel along the tree of directories, retrieve files by path and open/close them, other than doing explicit operation on files. How and when data is stored on a file and retrieved from it is completely up to the programmer, although some programming languages provide serialization mechanisms as a standard method to preserve language entities on a file.

It should be noted that files can also be used as a technique for inter-process communication (IPC) if the same file is opened my multiple processes. They can use it to communicate to each other, effectively letting data through the process barrier.

2.3 Serialization

Serialization[14] is the process of transforming data structures or object state into a linear stream of byte. The reverse operation of recreating the data structure or object from the serialized representation is called deserialization.

Serialization is commonly used to store objects on files so that they can be retrieved later and to perform inter-process communication. To be more precise, it can be used to transform data in a format which is suitable for communication, in order to distribute it among components of a system, and to implement remote procedure call (RPC). Thus, serialization is a mechanism valuable for persistence and distribution of data.

We can distinguish two kind of serialization based on the output format of the process: binary or text-based. The former it is in general more efficient

[14] In general the term *serialize* is considered to be synonymous with *marshal*, although in some context their meanings are slightly different.

in terms of memory space and time, whereas the latter is more portable and human readable. Example of well-known text-based serialization format are eXtensible Markup Language (XML) [Bray et al., 2008] and JavaScript Object Notation (JSON) [ECMA, 2013, Bray, 2014].

Serialization and deserialization must be coded explicitly by the programmer if the language's runtime does not support type introspection, for instance in C. Furthermore, the above-mentioned cases introduce interesting challenges: data consistency has to be ensured (for example the object graph should not contain dangling references), new type might need to be introduced at runtime and has to be ensured that a program can operate as expected on the deserialized objects (for example that they satisfy a certain interfaces). Because of these issues, serialization usually require some support from the runtime.

Languages that provide introspection and dynamic loading[15] usually offer automatic serialization as part of the standard library, for instance in Java [Oracle Corporation, 2005], or as an external library. In some cases additional annotations [Hericko et al., 2003, Cisternino et al., 2005, Tansey and Eli, 2008] are required in order to identify data to serialize or to change serialization format.

Languages that provide automatic serialization, in the programmer prospective, reduce in some way the effort required to implement persistence and distribution, but do not eliminate the explicit management necessary to store and retrieve data from files or to send and receive data through networks. In listings 2.5 and 2.6 two Java examples from [Oracle Corporation, 2005] are shown. The first one show how to serialize data on a file whereas the second one how to deserialize it.

```
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s = new ObjectOutputStream(f);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

Listing 2.5 A program that serialize today's date to a file.

In listing 2.6 is possible to see that explicit casting operations are required in order to recreating objects of the appropriate types. This additional steps are necessary due to the strong type system of Java, but in general the test of data type is indispensable to avoid runtime error and manage properly incorrect situations.

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
```

[15] More precisely, dynamic class loading in the context of object-oriented programming languages.

```
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Listing 2.6 A program that deserialize a string and date from a file.

It is worth noting that serialization, similarly to what happened in Orthogonal Persistence, acts on the entire transitive closure of the object to serialize. However, in this context the output stream of the process must be loaded or saved (sent or received) in a single operation. Furthermore, this technique cause the creation of a new copy of the data: in an object-oriented settings, "This breaks referential integrity since there is no way of matching the identity of objects from different save/load operations" [Dearle et al., 2009].

2.4 Databases

Databases are organised collection of data, which is arranged according to a data model in order to be easy to manage. The software that provides access to databases to users or other applications is the database management system (BDMS). It is designed to allow the definition, creation, querying, update, and administration of databases.

Databases have traditionally been designed for structured data, but over the years support for semi-structured (for instance XML an JSON) and unstructured (for example text, audio and video) data have been provided too. They offer higher level operations and, in general, more advanced features when compared to files, but the details depend on the particular data model adopted and the implementation characteristics.

In the next sections we briefly report different database models and highlight the main features that each one has. Our primary interest is how programming languages interact with databases.

2.4.1 Relational databases

Relational databases organise data according to the relational model introduced in [Codd, 1970]. In this model data is represented in terms of tuples, that are grouped into relations. Each tuple in a relation is univocally identifies by its primary key: a set of attributes of the tuple. A primary key can be used as foreign key in tuples of other relations resulting in conceptual links between tuples. This information is used to recombine data from different relations into a single collection.

Relational DBMSs (RDBMS) support transactions [Gray, 1981], which consist of a sequence of operations on a database that are executed in a reliable way ensuring ACID[16] properties [Härder and Reuter, 1983]. They have

[16] ACID stands for Atomicity, Consistency, Isolation and Durability.

SQL (Structured Query Language) as standard interface for data definition, data modification and query operations. SQL is indeed a special-purpose programming language that has to be used within a general-purpose programming language in order to define the interaction between an application and the database where data is preserved. This situation lead to the problem of how to use SQL from another programming language.

Embedded SQL is an approach to combine programming language with SQL that consist in adding SQL statements inline to program source code of the host language. This method required an additional step compared to the traditional compilation process: a dedicated preprocessor replaces SQL statements with host-language calls to a code library before the actual compilation. Despite this approach lets to validate SQL statements and apply other mechanisms at compilation time, it lacks the possibility to dynamic build SQL statements at runtime[17].

The most common method to interact with databases from a host language is through a set of APIs provided by an abstraction library. Examples are Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). These libraries offer APIs to connect to databases, define, execute and retrieve the result of SQL statements and allows the management of transactions. In this approach, SQL statements are expressed as strings in the host language and are validated at runtime: any error conditions are raised at runtime and should be properly managed. Moreover, programs that interact heavily with databases have to perform much string manipulation.

Several companies have taken another approach to build RDBMS. This approach consists in extending SQL with other programming language constructs in order to produce a general-purpose programming language that embed SQL by design. An example is PL/SQL, a procedural extension for SQL developed by Oracle Corporation for its relational database.

These three approaches are different from each other for how SQL is integrated in the host language but none of them reduce the amount of work that the programmer has to do to manipulate persistent data, which need to be explicitly retrieved and stored in the database. Furthermore, until now our discussion has concerned programming languages in general but has not discussed the specific problems that arise when the host language is object-oriented.

The conceptual and technical difficulties that arise when a relational database is used from a object-oriented programming language are referred to as *object-relational impedance mismatch*. The fact that an object encapsulates its state and behaviour (methods), whereas a tuple regards only state, is the fundamental aspect that distinguish between object-oriented paradigm and relational model. To overcome this problem in general only

[17] An example of embedded SQL is SQLJ, an outdated approach to combine Java and SQL.

state is preserved into databases, so encapsulation is violated[18]. Furthermore, relational databases do not support inheritance and polymorphism, other important object-oriented concepts. On the other hand, the concept of transaction and the related properties are not supported out of the box in a object-oriented language. From a technical prospective, the main problem are the differences between data types of relational databases and data types of language's type systems. Another obstacle is the different method to link information: relational databases use foreign keys to identify other tuples and join operations to reconstruct data, while object-oriented languages use direct references (pointer) to other objects[19]. Further information about this impedance mismatch can be found in [Ireland et al., 2009].

In order to free the programmer of the task of explicitly managing the mapping between two different type systems (the one of the database chosen for preserving data and the one enforced by the programming language used to develop an application) the technique of object-relational mapping (ORM) has been proposed. A mapping depend most on the specific language it targets, so a general treatment of the argument is out of our scope. What is interesting for us is the possibility that an object-relational mapping allow: using language features to program the interaction with a database.

As an example, we consider the Java language and two different ORM specifications that have been developed for it. The first and older one is Java Data Objects (JDO). It is both an object-relational mapping and a transparent object persistence standard, because it allows to used other data storage like XML files. For the programmer, a drawback of this specification is that the relationship between object and persistent data is specified in the external XML metafiles. The source code does not contain any information about persistent data, expect for API calls needed to store and retrieve data, but the specification must be defined in any case. JDO also provide a query language to abstract over the underlying storage technology: JDO Query Language (JDOQL).

The second and more recent ORM specification developed is Java Persistence Api (JPA) [Oracle Corporation, 2013]. In contrast to JDO, it is only a object-relational mapping but it eliminates the require of an XML mapping definition and allow to use annotations. As JDO, also JPA provide a query language: Java Persistence Query Languages (JPQL) that resembles to SQL but operates on JPA objects. JPA offers many annotations but only few are necessary, the other ones are optional and are needed to specify further details. Listings 2.7, 2.8 and 2.9 are examples taken from [Oracle Corporation, 2013]. In the first one a class Employee is defined and the `@Entity` annotation is used to specify that the class is a JPA entity and

[18] If a RDBMS also provide the functionality to store procedures, in any case state and code are stored separately.

[19] In other words RDBMS represent data in a tabular format, whereas object-oriented languages represent it as an interconnected graph of objects.

therefore a database table. The `@Id` annotation is used instead to specify fields whose database columns correspond to a primary key.

```
@Entity
public class Employee {
  @Id long empId;
  String empName;
  ...
}
```

Listing 2.7 JPA: define an entity class with a simple primary key.

The second example just shows how to define a transient field that will not preserved into the database using the `@Transient` annotation.

```
@Entity
public class Employee {
  @Id int id;
  @Transient User currentUser;
  ...
}
```

Listing 2.8 JPA: specify a not persistent field.

In the third and last example a method retrieves information from the database and then stores a new object into it, using the dedicated API. The interaction with the database is not direct, but is mediated by an `EntityManager` instance that is associated with a persistent context. The `EntityManager` API allow to manage the entity instance lifecycle. In this case the method `find` is used to retrieve a customer using its primary key, whereas `persist` lets to store and manage an order object. The `EntityManager` instance manages the persisted object, so future modification to the object will be tracked and the database will be updated.

```
@PersistenceContext EntityManager em;

public void enterOrder(int custID, Order newOrder) {
  Customer cust = em.find(Customer.class, custID);
  cust.getOrders().add(newOrder);
  newOrder.setCustomer(cust);
  em.persist(newOrder);
}
```

Listing 2.9 JPA: use of EntityManager for persisting an object.

Another interesting approach of using language features to interact with databases is the one introduced in .NET Framework 3.5, that is Language Integrated Query (LINQ) [Torgersen, 2006, Torgersen, 2007, Microsoft Corporation, 2013].

LINQ is a framework that adds native data querying capabilities to .NET languages and is meant to provide a uniform method for querying of data across different data source. It can be used not only for data storage such as databases and XML files, but also for in-memory collections.

This approach is a mix of two of the three methods described before to interact with databases from a language: it combines embedded SQL and dedicated APIs. On one hand, LINQ indeed can be understood as a sub-language that extends the language by the addition of query expressions. Furthermore, it looks very similar to SQL because many standard query operators have the same name and semantic of SQL statements or clauses. On the other hand, specific APIs are provided to create databases, insert objects into a database and to commit changes made.

Listing 2.10 shows a LINQ query in C# for retrieving customers from a database[20]. The result of the query is then accessed with the `foreach` statement, which let to iterate over the customers.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

IQueryable<Customer> custQuery =
  from cust in db.Customers
  where cust.City == "London"
  select cust;

foreach (Customer c in custQuery) {
  ...
}
```

Listing 2.10 LINQ: query in C#.

In the previous example, the so call "LINQ to SQL" framework is used. It consist of a "LINQ provider that implements the query operators by translating expression tree to SQL and sending them to a relational database" [Torgersen, 2006] and a full object-relational mapping for C#. Further details can be found in [Kulkarni et al., 2007]. It is worth noting that query like this one can be static type checked and optimised, but it is also possible to define query dynamically at runtime.

Listing 2.11 shows instead how to create a new customer and add it to the database using the dedicated API of LINQ.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

Customer cust = new Customer();
cust.CompanyName = "SomeCompany";
```

[20] Northwnd inherits from System.Data.Linq.DataContext, that represents the main entry point for the LINQ to SQL framework.

```
cust.City = "London";
cust.CustomerID = "98128";
cust.PostalCode = "55555";
cust.Phone = "555-555-5555";
db.Customers.InsertOnSubmit(cust);

db.SubmitChanges();
```

Listing 2.11 LINQ: persist an object in C#.

LINQ to SQL uses annotations of class definition in order to specify the details of the underlying database, like the JPA specification.

2.4.2 Object-oriented databases

Object-oriented databases adopt as data model the one proposed by the object-oriented paradigm and embrace its concepts. Thus, data is represented as objects into the database in the same way it is described in the programming language that manipulates it. This type of databases appeared in the mid 1980s [Maier et al., 1985, Derrett et al., 1985, Dittrich, 1986] as response to two shortcomings of the relational ones, as described in [Maier, 1989]. The first issue was that relational databases were not the most suitable solution for a certain kind of applications, for example computer-aided design (CAD). The second problem was the already mentioned object-relational impedance mismatch.

The fundamental features of an object-oriented database, together with other optional ones, have been described in "The Object-Oriented Database System Manifesto" [Atkinson et al., 1989]. Among these, a very important characteristics that an object-oriented database must have is the support of object identity, as it is also noted in [Dearle et al., 2009]. The paper states that identity "is perhaps the biggest differentiating feature between an OODB and a relational DB". This statement should be understood as conceptual difference: the relational model requires a primary key that unique identifies each tuple whereas objects have uniquely identities formed when they are created. Other issues related to object-oriented databases are discussed in the same paper.

There is no general method to retrieve objects from an object-oriented database and different approaches have been adopted. They can be grouped in two classes: ad-hoc and declarative methods. Among the former type we can mention ZODB [Zope Foundation, 2015], an object database for Python with a dictionary at the root. It allows to retrieve all the objects stored or a specific one which is bound to an identifier in the dictionary. The declarative approach instead has been influenced by SQL; examples are Object Query Language (OQL) and the query languages related to JDO and JPA already mentioned. Examples of queries of these and other similar query languages

can be found in [Takasaka, 2005] and [Cook and Rosenberger, 2005]. The latter also described an approach in which APIs are used to express query, called "native query".

Nowadays object-oriented database may be included in the broad definition of NoSQL databases, as noted in [Cattell, 2010].

2.4.3 NoSQL databases

NoSQL is commonly interpreted as "Not Only SQL" and it is used to indicate databases that may also support SQL-like query languages. However, in our view this term is referred to databases that adopt a non relational model, as in [Leavitt, 2010, Cattell, 2010, Huang and Luo, 2013]. Under the umbrella of NoSQL databases different data models are grouped, including key-value stores, document-oriented stores, column-oriented stores and graph databases. Compared with relational model, where it is mandatory to define a database schema before actually storing data, these models usually relax the requirement and allow to define the structure of data along with the data itself (for example in key-value stores) or to add a new attribute at any time (as in column-oriented stores).

There is no precise definition of what is a NoSQL database, due to the variety of approaches that are classified as NoSQL. Therefore, only a loose description of what are its main characteristics is usually given. According to [Cattell, 2010], NoSQL databases generally have six features:

- the ability of horizontally scale (scale out) "simple operations" throughput over many servers,
- the ability to replicate and to partition (shard) data over many servers,
- a simple call level interface or protocol,
- a weaker concurrency model than the ACID transactions of most RDBMS,
- use distributed indexes and RAM for data storage efficiently,
- the ability to dynamically add new attributes to data records.

The ability to scale out[21] is probably the core feature that a NoSQL database should have, since this kind of technological change has been triggered by the growing amount of data and number of users that companies such as Google and Amazon have to manage. This aspect is mentioned in [Leavitt, 2010, Cattell, 2010, Pokorný, 2011, Strauch, 2012, Huang and Luo, 2013] and some of these papers also identify Google's BigTable [Chang et al., 2006] and Amazon's Dynamo [DeCandia et al., 2007] as the earliest influential systems for the following NoSQL databases.

The "simple operations" to which Cattell refers are key lookups, reads and writes of one record or a small number of records. This kind of workload

[21] To scale out means to add more nodes to a system.

is traditionally called online transaction processing (OLTP) and it is the one for which NoSQL databases are most often considered[22], as also noted in [Stonebraker, 2010]. Operations like complex queries or joins are not common in this context and, in some cases, not even possible.

Replicating and partitioning data over many servers are techniques useful for horizontal scaling. In particular horizontal partitioning, also known as *sharding*, is a valuable method because it can improve both read and write performance. NoSQL systems usually adopt a shared nothing architecture[23] in which each node is independent and self-sufficient, on top of which automatic sharding may be provided. This is the capability of the system of taking on the responsibility of allocating data to shard (partition) and ensuring that data access goes to the right shard[24]. Shared nothing and automatic sharding are also emphasises in [Stonebraker, 2010] as a way to improve OLTP performance.

The adoption of a weaker concurrency model was pioneered by Dynamo that uses *eventual consistency* as consistency model in order to achieve higher availability and scalability. The same approach was proposed in [Pritchett, 2008] with the name of BASE, acronym for "Basically Available, Soft state, Eventually consistent", as an alternative to ACID. The idea is to relax the consistency requirement to promote availability. As a reason for that is often cited Brewer's CAP theorem, which we mentioned in Section 1.5. The same Brewer stated in [Brewer, 2012] that his theorem has led to a wide variety of novel distributed systems and that the NoSQL movement has applied it as an argument against traditional databases. Nevertheless, there are some NoSQL databases that provide ACID transactions.

In the following we briefly describe the data models adopted in NoSQL databases, mostly according to the classification presented in [Cattell, 2010]. Other categorisations of NoSQL databases have been proposed as described in [Strauch, 2012]. For each model, we outline the common aspects regarding data representation, data organisation and the interface provided. Further details and a description of existing databases can be found in [Cattell, 2010, Pokorný, 2011, Strauch, 2012, Huang and Luo, 2013]. The data models are:

- Key-value: The key-value data model is the simplest one and can be understood as a map or dictionary of data. Data is stored as value and an index is used to find it, based on a programmer-defined key. The type of a key depend on the particular system, but in some cases complex types can be used (for instance in Project Voldemort[25]). A value can be a string or blob, so a serialization algorithm must be

[22] This workload is common in modern web applications.

[23] This term first appeared in [Stonebraker, 1986] and such architecture has been used also in some RDBMS.

[24] One form of automatic sharding is consistent hashing.

[25] `http://www.project-voldemort.com/voldemort/`

used to transform data into a text or binary format. As an example, JSON is a text format usually supported. There are only few systems (like Redis) that also support more complex structures as value[26]. A characteristic common to all databases that adopt this data model is that values are opaque and can not be directly modified. We mean that to modify a value (as an example a JSON) the only possible way is to retrieve the value with the proper key, modify a part of it and then store the modified value with the same key. The interface provided to the programmer are in general API calls to insert values with a specified key, retrieve value by key and to delete key-value pair. Most of the system support only one index and does not support joins and aggregate operations, that must be properly coded by the programmer. Some system store keys in order and allow to retrieve ranges of them.

- Document-oriented: The document-oriented data model organises data in documents. A document in this context is a semi-structured data and let to store nested structures, lists of values and scalar values. Typical document formats are JSON and XML. In contrast to the key-value model, in this model documents are transparent and it is possible to search by attributes. It is worth noting that there is no strict schema documents have to conform to and attributes are simply names. Documents are organised in what we can call "collections", but the terminology is not standard and depends on the particular system. A collection is just a group of documents and it is the abstraction provided to the programmer to put together related documents[27]. Databases adopting this model provide a mechanism to query collection with the possibility to use multiple attributes value constraints. This query interface in some cases is string-based and similar to SQL but with some limitations; in other cases it is offered through specific calls. These system generally support secondary indices to speed up specific queries and provide an update operation to modify single attributes.

- BigTable-inspired: This data model represents data as multidimensional values, which are identified by a key. The terms commonly used are rows and columns: a row is what we have called multidimensional value whereas a column is an attribute of such value. Rows are organised in tables and columns are arranged in groups. This model is inspired by the one adopted in Google's BigTable [Chang et al., 2006][28] and has been named differently by various authors. For instance, in

[26] In Redis a value can also be a list of strings or a set of strings.

[27] Some system store different collections on different nodes, but this detail depend on the implementation.

[28] In this paper it is simply stated that "a table in Bigtable is a sparse, distributed, persistent multidimensional sorted map".

[Cattell, 2010] it is called "extensible record" but unfortunately this term tends to cause confusion with the concept of records in relational databases. In [Strauch, 2012] it is instead called "column-oriented", but the author underlines the his view is less puristic than the precise meaning, which refers to a model where data is stored by column, and subsumes model that integrate column and row orientation. The same paper also reports "wide columnar" and "entity-attribute-value" as names used by other authors. Finally, in [Sadalage and Fowler, 2012] this model is called "column-family" due to his notion of grouping[29]. Data is structured in order to be (first) partitioned horizontally, rows are split through sharding on the key, and (then) vertically, columns are organised in "column groups" that are treated separately. The column groups must be predefined, but new attributes (columns) can be defined at any time. Databases that adopt this data model generally provide an interface to insert and retrieve rows, insert, update and delete columns of a particular row and in some cases also MapReduce [Dean and Ghemawat, 2004] operations. A few systems offer this interface through a query language.

2.5 From persistence to distribution

In the previous sections we have reported various idea, and related implementations, to provide persistence in programming languages, either as a language concept or using an external system. However, we have focused only on one aspect of our problem and did not address the other: distribution.

Distribution is the movement of entities (data, objects, components, services, etc) between parts of a system and the coordination among them. This definition is on purpose general in order to include more system that at a certain abstraction level have similar characteristics: from multithreaded programs to distributed systems. All these systems are arranged in subparts representing autonomous computational entities that cooperate to perform a specific task or solve a given problem. They also provide a communication mechanism without which the distribution would not be possible.

There are two main classes of communication mechanisms: message passing and shared memory. The former consists in sending and receiving messages, whereas the latter involves writing and reading a shared memory[30]. Which mechanism a system implements depends on it characteristics; it is common to have a shared memory in multicore processors whereas in a distributed system the communication is made through messages. However, in

[29] In BigTable a group is called *column family*.

[30] In this case we have used well-known terms, but it would be more correct refer to these mechanism as entities passing and shared space to be more general.

general it is possible to build a shared memory abstraction on top of message passing system and vice versa. Thus, the choice of which mechanism a system provides can be done during its design.

An important property related to distribution is transparency, as pointed out in [Tanenbaum, 1993, Schwarzkopf et al., 2013]. We can talk about transparency in relation to both the movement of entities and the coordination of the parts.

On one hand, in systems adopting a message passing technique it is possible to have naming transparency if subparts are identified by (unique) names, rather than worry about their relative locations. Moreover, there are systems providing automatic dispatch: the subpart to which send a message is selected by the system. Such automated coordination represents another type of transparency for the user of the system. Finally, it also possible to have communication transparency if non-local actions are executed in the same manner of the local one.

On the other hand, shared memory abstraction provides a natural communication transparency due to the fact that the communication is implicit when a modification is made to the shared memory. In these system, much of the effort of the user concerns the coordination and the synchronisation of the subparts, although some systems provide mechanisms to automatise these aspects.

The most general kind of transparency is distribution transparency, which refers to the ability of a system to appear as a whole, even though it is divided into subparts. As an example, most modern databases are distributed system, especially NoSQL ones, and have this property: users interact with the system as it were one logical system. As we have mentioned in Section 2.4, the distribution of a database among different nodes is a useful technique to improve availability, reliability and performance. Despite the fact that such property is valuable, it is not achievable or even desirable in all systems.

In the following we briefly report on a few systems that provide distribution, taking into account only data. Also in this case we adopt a programming language prospective, so our interest is in how distribution is offered in languages.

2.6 Distribution within programming languages

Programming languages have initially been designed to write sequential programs that would be run on a single processing unit. Although starting from the 1980s a certain number of concurrent programming languages have been proposed and the widespread adoption of multicore processors since early 2000s, well-known and widely used languages have provided only sequential programming until recently.

The most well-known example of this is C, that was defined as a sequential programming language until the C11 standard revision. Before that, to take advantage of the multithreaded underlying machine and develop a concurrent program an external library that provides threading primitives had to be used. [Boehm, 2005] shows which are the limitation of the approach initially taken in C, that was replaced in C11 with the introduction of a detailed memory model and the support for multiple threads.

We can recognise a similar situation in how distribution within an operating system or across a network is made in programming languages. Although concurrent programming languages provide different unit of executions and a communication mechanism to organise the structure of a program, the interaction is usually limited to the execution boundaries, traditionally the process. Communication between two processes are made through an interprocess communication (IPC) mechanism, which can be used within the operating system or across the network. Indeed, almost all programming languages which have IPC mechanisms, provide them through a library that adds a distribution layer on top of the existing language. Examples of this are networking libraries that provide socket abstraction or libraries that implement the message passing interface (MPI).

The choice of providing communication and distribution in an object-oriented language through a library has some disadvantages. Among these, the most problematic is the fact that the process of transformation from the internal representation of an object to a linear one suitable to be transmitted causes the loss of the identity of the object. What is transferred is indeed a value without identity: the semantic information related to the value is not transmitted. This problem is similar to the one that usually happens when an object is persisted.

Our research of languages that have been designed considering distribution mechanism has led the following results: Emerald, Orca, Obliq and Distributed Oz. Although all these languages have been proposed before 2000 and so designed in a different scenario compared with actual one, they represent interesting solution to the problem of how integrate distribution into a language.

Emerald [Black et al., 1986, Black et al., 1987, Jul et al., 1988] is an object-based programming language for the implementation of distributed applications in local area network (LAN) of independent nodes. Like Smalltalk-80, Emerald considers all entities to be object. Unlike Smalltalk-80, it is a strongly typed language and has no classes or inheritance. Objects can either be passive (data) or active (contains a process). The language has an explicit notion of object location[31] and mobility, so an object can be moved from a location to another. In [Jul et al., 1988] this kind of mobility is called fine-grained and it is proposed as an alternative to process mobil-

[31] At any given time, an object is on a single processor, called its location.

ity. Another interesting characteristic of Emerald is that each object has an identity, which distinguishes it form all other object within the network. In details, to "each remotely accessible object is assigned a unique object id (OID)" and "each node has an object table that stores pointers to the local object descriptors for all remotely accessible object" [Black et al., 1987]. Moreover, object invocation is location independent (or transparent, as we have called it in the previous section), is in the responsibility of the system to locate the target of the invocation.

Orca [Bal et al., 1990, Bal et al., 1992] is a procedural parallel programming language intended for implementing parallel applications on distributed systems. It supports a communication model based on shared data, but this sharing data is logical rather than physical since distributed systems lack shared memory. The parallelism is based on sequential processes and the language provides an explicit fork primitives for spawning a new child process and passing parameters to it. Sharing of data happens on process creation, but has to be specified in the declaration of the child process (shared parameters). The language imposes a limitation on shared data: only user-defined abstract data types can be shared. As far as the implementation is concerned, in a prototype implementation of Orca shared data is replicated on all processors and a ordered broadcast protocol is used for updating copies.

Obliq [Cardelli, 1994, Cardelli, 1995] is a lexically-scoped interpreted language that supports distributed object-oriented computation. The lexical scoping in this context, due to the distributed characteristic of the language, assume an additional meaning compared to the classical one: "it ensures that computations have precise meaning even when they migrate over the network: a meaning that is determined by the binding location and network site of identifiers, and not by execution sites" [Cardelli, 1995]. Obliq is based on objects and Cardelli described it as an embedding-based language, such name indicates that all methods on an object, as well as its value fields, are embedded in the object itself (he further specify at least in principle) rather than being located in other object or classes. The explanation of this choice is that the self-contained nature of the object is well suited to network applications. The distributed semantic of the language is based on the notions of sites, locations, values and threads. Sites are address spaces and contain locations, and locations contain values. Threads are virtual sequential processors. Values include basic values, object, arrays and closures. A values may contain embedded locations, as example an object value has embedded locations for its fields. Values may be transmitted over the network: values containing no embedded locations are copied, whereas values containing embedded locations are copied up to the point where these locations appears; local references to locations are replaced by network references. Thus, sending an object causes the creation of network references for its fields[32]. In

[32]We can think to the identity of an object as the combination of the locations of

Obliq every object is bound to a unique site (the one where it has been created) and does not move. However, objects can be cloned to different sites and, furthermore, this operation can be combined to aliasing to achieve migration: cloning provides state duplication and aliasing redirects operations to the clones. More migrations of the same object cause the creation of a chain of indirection. To avoid this problem and allow the creation of the initial network reference to an object on another site, the language considers an external process that acts as name server, which store the association between strings (names) and network references.

Distributed Oz [Haridi et al., 1997, Roy et al., 1997] is an extension of Oz, a concurrent language with first-class procedures, with two concepts: mobility control and asynchronous ordered communication. It is in some sense inspired by Obliq, from which takes the distinction between values and locations. The language distinguishes between variables, records[33], procedures, cells and ports. All stateless entities, namely records, procedures and variables[34], are replicated (copied to a site). Instead, cells and ports are stateful entities and are subject to the mobility control, in other words the ability to migrate between sites or to remain stationary at one sites, according to the programmer's intention. In both the above-mentioned papers, cells are mobile and ports are stationary. A cell is a pair representing the mutable binding of a name to a variable that, in addition to an state-update operation, provides an exchange operation which causes the state to move automatically to the site invoking it. On the other hand, a port is a pair of an identifier and a stream and provide two operations: send, which append the entry to the stream in an asynchronous ordered manner, and locales, which causes the port's state to move automatically to the site invoking it.

2.7 Distributed operating systems

A distributed operating system is an operating system over a collection of independent, autonomous, communicating computers that appear to the users of the system as a single computer [Tanenbaum, 1993, Tanenbaum, 1995].

Thus, a distributed operating system is a layer of software that manages the underlying hardware and provides an environment in which a user can execute programs in a convenient manner, as a traditional (centralised) operating system. However, it also create the illusion in the minds of the users that the entire network of computer is a single system, rather than a collection of distinct machines[35].

its fields, even though in the implementation, as point out in [Cardelli, 1995], network references are generated for objects and not for each of their embedded locations.

[33]Under the name of *record*, in [Haridi et al., 1997], is grouped together common data types (records, number, strings, etc).

[34]Oz variables are single-assignment variables or more appropriately logic variables.

[35]Some authors refer to this property as the single-system image, as noted in

In Section 2.2 we have mentioned files as a mechanism, provided by the file system, to persist data. In a distributed operating system such solution could allow to have both persistence and distribution, due to the characteristics of the operating system. Indeed, a distributed operating system has a file system that must look the same from every computer in the collection. Furthermore, as specified in [Tanenbaum, 1995], every file should be visible at every location (according to protection and security constraints).

As an example, in this environment a process can save data on a file that can later be read from another process to retrieve the information. These processes can run on different computers but any of them use the file as if it were local to the computer where the process run; it is in the responsibility of the system to ensure that processes can have access to the file (by moving or replicating it on the other computer, for instance). However, this kind of distribution is made at the file system level.

Another possibility to have (implicit) distribution is that the collection of computers shares a single virtual address space. However, this solution was not immediately examined. As described in [Tanenbaum, 1995], in the early days of distributed computing, everyone implicitly assumed that programs on machines with no physically shared memory obviously ran in different address spaces and communicated by message passing. A shared memory model, also known as distributed shared memory, was proposed in [Li, 1986] (and later also described in [Li and Hudak, 1989]). It operates as a paging system across machine boundaries.

The system proposed by Li is well described in [Tanenbaum, 1995]: "in essence, this design is similar to traditional virtual memory system: when a process touches a nonresident page, a trap occurs and the operating system fetch the pages and maps it in. The difference here is that instead of getting the page from the disk, the operating system gets it from another processor over the network. To the user processes, however, the system looks very much like a traditional multiprocessor, with multiple processes free to read and write the shared memory at will."

Unfortunately, despite being easy to program, this system exhibits poor performance for many applications. Various approaches to make distributed shared memory more efficient have been attempted, as an example one of them does not share the entire address space, "only a selected portion of it, namely just those variables or data structures that need to be used by more than one process" [Tanenbaum, 1995].

Is out of our scope to report the different aspects and issues that regards distributed operating systems. For a complete treatment of the subject we refer to [Tanenbaum, 1995], which describes communication and synchronisation among other things.

We have instead recalled a few ideas, developed in the research area of

[Tanenbaum, 1993, Tanenbaum, 1995].

distributed operating systems, that in our opinion are interesting also nowadays and should be investigated. Indeed, this research topic had the most successful period in the 1980s, while most recently the interest about this argument has been only moderate. However, we are not the only one interested in distributed operating system, as an example [Schwarzkopf et al., 2013] argues that the distributed OS concept is worth a revisit.

This paper discusses distributed operating system in the context of data centers, but reports considerations valuable both in general and in our specific scenario. First of all, it indicates three motivations of the original lack of success of distributed OS, namely:

- the cost of transparency, that in some cases turned out to be a hindrance to deterministic performance or led to poor performance;
- the compute/communication speed dichotomy, because clock speed increased more rapidly than communication speeds causing the high time cost of remote operations;
- the conflation of micro-kernels and distributed system, because many distributed OS research project chose a micro-kernel as a central design component that put them at a further disadvantage in the competitive arena of 1990s desktop computing, as they also inherited many perceived drawbacks of the micro-kernel approach (for instance in performance).

Then, it mentions some key conditions that have recently changed, as example that I/O is faster than computation, data center applications necessitate distribution and transparency is in demand. We can understand this last condition in the general scenario of distributed applications, recalled in Section 1.3, and affirm that the programmer need to be liberated from knowing the details of distributed coordination. Finally, the authors motivate why the time is right to reconsider the distributed OS concept and outline, among the others things, the possibility to have automated decisions and the opportunities for new abstraction.

The paper also discusses some key concept and challenges. Most notably, in our view, it states that might be worthy to investigate the concept of unified naming and distributed shared memory. Furthermore, it mentions weaker consistency models for higher availability, which have recently received much attention, and suggests to support them at the operating system level.

2.8 Eventual consistency and conflict resolution

Replication is a technique used in distributed systems to guarantee consistent performance and high availability, particularly in those system that operate on a worldwide scale [Vogels, 2008]. However, those systems try to

make replication as much transparent as possible and to appear to the user as a whole: that requires maintaining the shared state consistent through replicas.

The CAP theorem concerns distributed systems, as already mentioned in Section 1.5, and can be concisely expressed as the fact that strong consistency (C) conflicts with availability (A) and partition-tolerance (P), or by the oft-used "two out of three" concept. However, this concept can be misleading as explained in [Brewer, 2012].

There are essentially three reasons why the "two out of three" concept can be misleading. First, because partition are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A occur many times within the same system at very fine granularity (for instance the choice can depend on the operation). Finally, all three properties are more continuous than binary. The author then explains that if there is a partition a decision must be made: consistency or availability. Furthermore, he affirms that in practice latency and partition are deeply related and when a timeout takes place during a communication, the partition decision must be done: cancel the operation or proceed with it. In a pragmatic view, a partition is a time bound on communication.

Distributed systems usually favour availability over consistency and adopt weaker consistency models. The most widespread model used is probably *eventual consistency*, a weaker form of consistency in which replicas eventually reach the same final state if clients stop submitting updates. In this model there may be replicas with stale state but, in the absence of updates, at a some point in the future that will be recovered. Examples of system that adopt this model are Amazon's Dynamo and other NoSQL databases.

In [Brewer, 2012] is also discussed the issue of partition recovery. It concerns how conflicting states, reached during the partition, should be merged. A similar problem exists when an update is executed locally at some replica and then sent asynchronously to the other replicas. In this situation concurrent updates may conflict, which means that a combination of updates (which may be individually correct) taken together can violate some invariant. To resolve conflicts, a consensus and a roll-back may be required. Since conflict resolution is hard, many system adopt ad-hoc approaches that can result error-prone.

The closest approach to a general framework for automatic state convergence is using commutative operations. This idea has been adopted in the development of conflict-free replicated data types (CRDT) [Shapiro et al., 2011c, Shapiro et al., 2011a, Shapiro et al., 2011b], a class of data structures that provably converge to a correct common state.

[Shapiro et al., 2011c] reports the formalisation of eventual consistency, the definition of the theoretically-sound approach proposed by Shapiro and colleagues, namely Strong Eventual Consistency (SEC), and the definition of CRDT. In the following we outline this work.

2.8.1 System model

Shapiro and colleagues consider a system of processes interconnected by an asynchronous network. The network can partition and recover. They assume a finite set $\Pi = \{p_0, \ldots, p_{n-1}\}$ of non-byzantine processes[36]. Processes in Π may crash silently; a crash replicas may remain crashed forever, or may recover with its memory intact. A non-crashed process is said correct.

A process may store objects. Here an object is a mutable, replicated data type that has an identity, a content (called its payload), and initial state and an interface consisting of operations. Two object having the same identity but located in different processes are called replicas of one another. With no loss of generality, the authors consider a single object with one replica at each process and, for simplicity, they assume a fully connected graph.

The environment consist of unspecified clients that query and modify object state by calling operations in its interface, against a replica of their choice called source replica. A query executes locally. An update has two phase: first, the client calls the operation at the source, then the update is transmitted asynchronously to all replicas. Two styles of replication are possible: state-based and operation-based (op-based for short).

In state-based (or passive) replication, an update occurs entirely at the source, then propagates by transmitting the modified payload between replicas. In contrast, in operation-based (or active) replication, the system transmits operations. [Shapiro et al., 2011c] describes in details both styles, here we consider only the operation-based replication because the payload transmitted in this approach is generally smaller than in the other approach and should be preferred to reduce bandwidth usage[37]. Nevertheless, some result about state-based replication will be mentioned.

An op-based object is a tuple (S, s^0, q, t, u, P), where S is the state domain and s^0 is the initial state. The replica at process p_i has state $s_i \in S$. A client of the object may read the state of the object via query method q and modify it via update method. However, in the op-based view an update is split into a pair (t, u), where t is a side-effect-free prepare-update method (for instance, it may compute result) and u is an effect-update method. The prepare-update executes at the single replica where the operation is invoked and it is followed immediately by effect-update method (to ensure causality between successive update).

The effect-update method executes at all replicas (said downstream). The source replica delivers the effect-update to downstream replicas using a communication protocol specified by the delivery relation P. That is, effect-update method u is enabled only if the precondition is satisfied. The delivery of u at replica i may be delayed, until $P(s_i, u)$ is true.

[36] All processes observe the same symptom if a fault occurs.

[37] The op-based approach is adopted in most popular collaborative software such as Google Docs, Google's collaborative word processor.

A method whose precondition is satisfied is said enabled (for instance, a non-null precondition may be necessary in some cases). It is assumed that an enabled method executes as soon as it is invoked. Method execution at replica i will be noted $f_i^k(a)$, where f is either q, t or u, and a denotes the arguments. The ordinal of execution f at replica i is noted as $K_i(f)$, i.e., $K_i(f_j^k(a)) = k$ for $i = j$, and is undefined otherwise. The states of a replica are numbered sequentially increasing with each method execution. A transition is noted $s_i^{k-1} \bullet f_j^k(a) = s_i^k$.

State equivalence $s \equiv s'$ is defined if all queries return the same result for s and s'. Both queries and prepare-update methods are side-effect-free, i.e., $s \bullet q \equiv s \bullet t \equiv s$.

Definition 1 (**Casual History (op-based)**). *An object's causal history $C = \{c_0, \ldots, c_{n-1}\}$ (where c_i goes through a sequence of states $c_i^0, \ldots, c_i^k, \ldots$) is defined as follows. Initially, $c_i^0 = \oslash$, for all i. If the k^{th} method execution at i is:*

- *a query q or a prepare-update t, the casual history does not change, i.e., $c_i^k = c_i^{k-1}$;*
- *an effect-update $u_i^k(a)$, then $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$.*

An update is said delivered at a replica when the update is included in the replica's casual history. Update (t, u) happenedbefore (t', u') iff the former is delivered when the latter executes: $(t, u) \rightarrow (t', u') \Leftrightarrow u \in c_i^{k-1}$, where t' executes at p_j and $k = K_j(t')$. Updates are concurrent if neither happened-before the other: $u \parallel u' \stackrel{\text{def}}{=} u \not\rightarrow u' \wedge u' \not\rightarrow u$.

A similar formalisation exists for state-based object; here we recall only that they have a method m (merge) that is used by a replica to merge its local state with the one received from a remote replica.

2.8.2 Strong Eventual Consistency

Eventual Consistency is formally defined as:

Definition 2 (**Eventual Consistency (EC)**).
Eventual delivery: *An update delivered at some correct replica is eventually delivered to all correct replicas: $\forall i, j : f \in c_i \Rightarrow \Diamond f \in c_j$.*
Convergence: *Correct replicas that have delivered the same updates eventually reach equivalent state: $\forall i, j : \Box c_i = c_j \Rightarrow \Diamond\Box s_i \equiv s_j$.*
Termination: *All method executions terminate.*

To avoid using roll-back and consensus, typically adopted in EC system, the authors require a stronger condition:

Definition 3 (**Strong Eventual Consistency (SEC)**). *An object is Strongly Eventually Consistent if it is Eventually Consistent and:*

Strong Convergence: *Correct replicas that have delivered the same updates have equivalent state:* $\forall i, j : c_i = c_j \Rightarrow s_i \equiv s_j$.

The intuition is to have replicas with the same (casual) history ensure having the same state, roughly speaking, without other mechanism.

Given that, the problem is finding a sufficient condition for objects in both state-based and op-based style.

2.8.3 Conflict-free replicated data types

To have conflict-free objects, the idea is to leverage simple mathematical properties that ensure absence of conflict. For example, in the case of op-based objects, we need the commutativity property for update operations.

Nevertheless, for op-based replication an additional assumption is made: an underlying reliable causally-ordered broadcast communication protocol, i.e., one that delivers every message to every recipient exactly once and in an order consistent with happened-before. Given this, it follows that two updates that are related by happened-before execute at all replicas in the same sequential order: $(t, u) \rightarrow (t', u') \Rightarrow \forall i, K_i(u) < K_i(u')$. However, concurrent updates may be delivered in any order. A sufficient condition for convergence of an op-based object is that all its concurrent operations commute.

Definition 4 (Commutativity). *Updates* (t, u) *and* (t', u') *commute, if and only if for any reachable replica state* s *where both* u *and* u' *are enabled,* u *(respectively* u'*) remains enabled in state* $s \bullet u'$ *(respectively* $s \bullet u$*), and* $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$.

An object satisfying this condition is called a Commutative Replicated Data Type (CmRDT).

However, the delivery precondition P may delay the delivery of an effect-update to some replicas. Therefore, for liveness, it must be proved that delivery is eventually enabled. For that reason the scope of the precondition is restricted to the ones for which causally-ordered broadcast is sufficient.

Theorem 1 (Commutative Replicated Data Type (CmRDT)). *Assuming causal delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition is satisfied by casual delivery, is SEC.*

The proof is presented in [Shapiro et al., 2011b].

As far as state-based objects are concerned, a similar result is presented in [Shapiro et al., 2011c]. In this case the sufficient condition for strong converge is that a state-based object is a monotonic join semilattice. In other

words, the set of states, equipped with partial order $\leq$, forms a join semi-lattice[38], merge computes the least upper bound of two states and updates monotonically advance upwards according to $\leq$. An object satisfying this condition is called Convergent Replicated Data Type (CvRDT).

Shapiro and colleagues have also proven that CmRDT and CvRDT are equivalent; in [Shapiro et al., 2011c] two results are presented proving that SEC op-based object can be emulated by a SEC state-based object and vice versa.

An example of CRDT is the PN-Set (Positive-Negative Set), in which to each element is associated a counter, initially at 0. Adding an element increments the associated counter, and removing an element decrements it. The element is considered in the set if its counter is strictly positive. However, due to the fact the a CRDT Counter can go positive or negative, it may happen that adding an element whose counter is already negative has no (visible) effect. This construction is a CRDT because it combines two CRDTs, a Set and a Counter.

This and other basic CRDTs can be found, together with other references, in [Shapiro et al., 2011a].

2.8.4 Remarks

CRDTs can be a valuable class of data structures to use in a distributed eventually consist system, and should be used when possible because they favour availability and ensure that state converges automatically.

Nevertheless, there are cases where a data structure can not be designed to be a CRDT. Indeed, from a technical point of view, CRDTs allow only locally verifiable invariants. To maintain a global invariant synchronisation is required[39].

[38] A join-semilattice is a partially ordered set that has a least upper bound for any finite subset.

[39] It is possible to enforce a local invariant that implies the global invariant, but this solution may be too strong.

3

Spray programming

> *"A possible first step in the research program is 1700 doctoral theses called 'A Correspondence between x and Church's λ-notation.'"*
> – Peter J. Landin, *The Next 700 Programming Languages*

This chapter presents our solution to the problem of providing persistence and distribution as programming language features, as proposed in Section 1.4. Our approach is to make a part of the heap persistent and distributed. We not introduce a new programming language that supports this idea, instead we augmented a programming language with a paradigm that deals with both persistence and distribution, that we call *Spray* programming. It has been designed to be implemented in existing programming languages extending them.

Firstly, we outline the context in which our solution can be applied and an informal description of it. Then, we discuss some design choices made in the early stages of development along with their motivations. Finally we present the specification of *Spray* using the Abstract State Machine (ASM) formalism.

3.1 Description

We consider a distributed system and a set of applications developed on top of it. Each of these applications can have zero or more processes that run on different devices, at different times and execute the specific application. We do not assume any particular architecture for the distributed system and related applications. We just consider processes interconnected by an asynchronous network that communicate exchanging messages. Moreover, we imagine a group of users that make use of one or more applications.

As an example, we can consider the World Wide Web (or simply the Web) and web applications. It is quite common to think at the same web app running on different web browsers, which are used by various users. The

state of each instance of the application depends on the particular user and other aspects related to the device.

A web application runs in a web browser but its code and data is stored on one or more servers. After having downloaded the client-side application's code, the browser sends request to the server to retrieve the data needed or to check for updates. Local changes of the application's state are also transmitted to update the remote copy of the state. This behaviour is typical of modern web application and must be coded explicitly by the programmer, together with other parts of the program. However, the task of keeping up-to-date the state of an application distracts from the business logic. This scenario gets more difficult if we consider that information may be shared among different applications of the same user or multiple users of one application.

Interesting examples of web applications are the ones in Google's or Microsoft's apps ecosystem. These ecosystems provide different types of web applications, including an email client, a time-management tool (calendar), a contact management tool, an office suite and a storage service . Some level of integration among the various applications is usually offered to the user which can access data of one application from another one. For instance, it is possible to see the list of contacts when writing a new mail.

Users of these ecosystem can also share documents, notes or events. A shared object represents a part of a web application that is shared among more users and allows them to interact. The kind of interaction depend on the specific application, for example a web-based word processor may provide collaborative editing of shared documents.

Moreover, the already mentioned app ecosystems usually have a mobile version of most of the web applications they provide. Thus, our considerations are not limited to web application but could be extended to mobile applications as well.

We now outline the basic traits of the applications we consider, of which web applications are a proper example. First, these application require the programmer to explicitly manage the back and forth of data between the local process and the remote ones, just to keep up-to-date the state of an application. Then, data is not only transmitted, but it is also stored both in the local device's memory and in remote storages to preserve the application's state beyond the execution of a single process. Pure web applications represent an extreme case, where all data is stored remotely.

If it is possible to store data locally; the programmer has the additional task to save and retrieve data from the local storage. In this scenario, the programmer has to deal with both the persistence and the transmission of data. Finally, since data is distributed among various processes, which can execute the same application or different ones, sharing of data should be considered.

We think that if persistence and distribution are provided as language

features, the programmer will be released from the burden of saving and transmitting data, together with the task of transform data in a format suitable for these operations. The runtime of the language may take care of these operations, while the programmer may focus on the business logic.

The main idea of *Spray* is to have a part of the heap which lifetime is detached from the process's lifetime. It models a memory that is persisted and distributed among more processes. This idea may be understood as a combination of Orthogonal Persistence (Section 2.1) and distributed shared memory (Section 2.7). Nevertheless, in the following discussion, we will refer to state when we talk about persistence and distribution, not simply data. In this context, a "state" is data together with the related semantic information given by the language and the programmer. The state of an application consists of the states of all its objects.

As already stated, we restrict our interest to object-oriented programming languages. This choice is not limiting, from a technological point of view, because most modern programming languages support the object-oriented paradigm and most of the application we consider, such as web and mobile applications, are written in object-oriented or object-based[1] languages. On the other hand, objects are a general abstraction to model a problem and its solution, and have the notion of identity. An identifier identifies an object during its lifetime and so may be used to refer to the object at different times and from various places. In other words, the same object can be pointed by different processes of the same application on the same device as well as by different processes of various application on distinct devices.

As far as the persistent and distributed memory is concerned, it is obvious that a flat organisation is not the right choice. Indeed, this solution implemented without other additional mechanisms may allow all processes to get access to the whole memory. This kind of approach is not desirable because the entire state of the memory is shared, whereas we want to let the programmer express which states are shared and which instead are not. In *Spray* the idea is to arrange part of the heap in a context of persistence called *scope*.

A *scope* is a special object that defines an environment and a memory. The scope's memory stores objects, which are referred by their identity, while the environment contains binding between local names and stored objects. Although the definition of scope in programming languages regards the visibility of a name binding, the part of a program where that binding is valid, in our discussion a scope determines the part of an execution where a name or an object is visible and therefore may be used. Instead of lexical scoping, we adopt dynamic scoping controlled by the programmer, who explicitly opens or closes a scope.

It is possible to store a scope inside another one since a scope is an

[1]Languages that support only some aspects of the object-oriented paradigm.

object. This leads to the possibility of storing nested scopes and retrieve them when necessary. However, there is a different between storing scopes and other objects: storing the same generic objects in different scopes at different times means to save a different version of it, whereas storing a scope means simply to store its identity in order to be able to retrieve it later. Thus, scopes can be used to obtain objects versioning, but not scope versioning.

The scope just described is called "single" scope. It is commonly what a programmer needs in the development of an application. As an example, the scope related to the application is used to store all the objects that the application manipulates. However, in some cases the programmer may want to express that an object is accessible only in a certain condition. For instance, he or she wants that a particular object of the application can be reachable only on a specific device. What he or she needs is method to express that an object is visible only if both the application and the device scopes are open. In order to let the programmer does this, *Spray* provides a special kind of scope called *allOf*.

This particular scope is obtained by viewing a set of scopes as one scope. As a single scope, it has a memory and an environment too. The aim of an *allOf* scope is to have a distinct scope that is open only if all the scopes that define it are open. The definition of this kind of scope is implicit, as well as its opening and closing. Furthermore, names and objects contained in an *allOf* scope should be visible in each scope that determines it.

If we do not take into account *allOf* scopes, *Spray* memory is conceptually organised as a hierarchy and the *root* scope is the top-most scope of such structure, from which all other scopes and objects can be retrieved.

Before moving forward to other aspects of *Spray*, it is worth noting that the description given for scopes is quite concrete, since a scope is explained as a container. Another possible and more abstract definition describes a scope as a label. Objects contained in a scope are simply objects with that label and names bound to an object are just decorations of the labelling. Nevertheless, we not delve deeper into this point of view here[2].

A unique feature of *Spray*, compared to other approaches to persistence, is the possibility of split an object among more scopes and recombine it when all those scopes are open again. The idea is not to persist the entire sub-graph reachable form an object, but to save just the first level replacing other object references with their identities. In other words, *Spray* adopts a shallow technique to persistence. This characteristic also allows to assign different objects to the same field of an object and to resolve which one is referred relying on what are the open scopes of a process at a certain time.

Spray memory is distributed among more processes and so there is the

[2]Formalising *Spray* according to this view may be an interesting investigation for further work.

problem of keeping up-to-date the memory's state in all processes. Taken into consideration that such system is a distributed system and so the CAP Theorem is applicable, we have chosen to favour availability over consistency. Thus, *Spray* adopts an eventual consistency model where each process has a replica of (a subparts of) the persistent storage and will eventually be notified of an update. This choice allows each process to apply a change locally without synchronization, and then send it to all other replicas.

Updates of persistent distributed objects are propagated to the programmer by the well-known observer pattern. The programmer specifies for which objects or fields of an object she/he wants to be notified and the runtime will take care of notifying him. *Spray* adopts a publish-subscribe pattern where callbacks are registered for events related to persistent objects or scopes. These callbacks are invoked later when the related event occurs on the local storage.

On top of the core of *Spray* programming, we assume a layer which has the task of retrieving scopes with a specific semantic meaning and binds them to predefined names. We can imagine various scopes with a precise intent, such as the scope of the current month, but we require at least three of them: user's scope, application's scope and device's scope. These scopes are special, and therefore essential, because they are related to a part of the system. For instance, an application's scope should be used to persist states of the application so that it is accessible to all the processes that constitute that particular application, no matter where and when they are running. In contrast, a device's scope is local to the device and can not be accessed elsewhere else. Only the processes running on that device may have access to its scope.

We recognise in these three scopes the fundamental traits of the modern applications mentioned before and believe that they are essential to express different kind of sharing, including sharing states related to a user on different devices or among various applications, as well as sharing states of an application between two or more users.

3.2 Design choices

In this section we explain more in details the fundamental choices made during the design of *Spray* programming.

3.2.1 Language level

Spray is designed to be integrated in a programming language and extends its runtime. This decision has been made to provide the programmer with an abstraction to manage persistence and distribution together, i.e., the scope.

Integrating scopes with other parts of the language avoids mismatches between the internal data representation and the external one suitable to

be persisted and distributed. It also shifts the task of translating a representation into another one from the programmer to the runtime. This transformation must be hidden to the programmer.

Furthermore, this approach allows to extend how objects are manipulated internally, yet maintaining the same interface for the programmer. In other words, transient and persistent objects can be used in the same way by the programmer while the runtime takes care of applying the additional operations needed for persistent objects.

In *Spray* we presuppose to be able to uniquely identify each object, no matter in which scopes it is persisted. Nevertheless, objects in programming languages are traditionally identified by their memory addresses and so their identity is guaranteed to be unique only within the process boundaries. Since we act at the language level, we required that objects have an identity unique among different processes running on distinct devices.

This can be obtained in traditional class-based programming languages without modifying the language itself by adding an additional preprocessing step that extends each class with a field for object's identity. In a similar way, methods for manipulating persistent distributed objects may be added or expanded in the output program[3].

3.2.2 Unique identities

The uniqueness of the object's identity in a distributed system such *Spray* is not a trivial property. We have to prove that this requirement is feasible. Furthermore, *Spray* demands that a new identity should be obtained without a central coordination that could result in a single point of failure.

Universally unique identifier (UUID), which is described in [Leach et al., 2005, International Telecommunication Union, 2012], represents a solution to this problem. An UUID is "an identifier that is unique across both space and time, with respect to the space of all UUIDs" [Leach et al., 2005], and has a fixed sized (128 bits). There are five versions of UUIDs that differ from each other for how an identifier is generated.

Although an UUID generator does not guarantee uniqueness due to the fixed sized and because generating the same identifier is possible, and so having a form of hash collision, in practice an UUID is reasonably unique and UUIDs have been used. For example, Java provides an UUID class in the standard library from its fifth version.

Microsoft has adopted the UUID standard as globally unique identifier (GUID). GUIDs are used in Microsoft's COM (Component Object Model) to internally identify the classes and interfaces of objects. Furthermore, COM is the basis for DCOM (Distributed COM), a technology for communication among software components distributed across networked computers.

[3] This may be implemented by making each class a subclass of the persistent distributed class, where are defined fields and methods for persistent distributed objects.

3.2.3 Shallow persistence

Spray adopts a shallow mechanism to achieve persistence that we call *shallow persistence*. This approach is in contrast with the one adopted by the systems reported in chapter 2, called "persistence by reachability" in the Orthogonal Persistence methodology.

Persistence by reachability means that a root object is persisted together with all the sub-graph it refers. We call this solution deep persistence to distinguish it from the one adopted in *Spray*, which instead preserves only the state of the root object. As already noted, references to other objects are replaced by their identities, while those objects are not preserved.

It is clear the shallow persistence is a more general solution compared to its deeper version. Indeed, with shallow persistence we may preserved all the sub-graph of an object by doing a recursive visit of the graph (cycles must be properly handled). In addition, shallow persistence lets to split an object in more parts that can be persisted in different scopes. However, the ability of persisting an entire object in parts required a look-up for the internal referred objects when the object is retrieved from a scope.

3.2.4 Eventual consistency

Spray adopts eventual consistency as consistency model. This choice has been made for two reasons. The former is that we are more concerned about availability than consistency. We do not want to impose any synchronization mechanism for every operations on a persistent object that a process can do. We would rather perform operations locally and then send them to the other processes of the system. This decision also imporves the user experience.

The latter motivation is that we will have the possibility in the future to extend *Spray* in order to manage offline operations. It may be also possible to notify the programmer when a partition is detached in order to allow him/her to disable some operations of the application[4].

An eventual consistency model introduces the issue of conflicts resolution for concurrent operations on the application's state. *Spray* avoids ad-hoc approaches for conflicts and advises against solutions where the responsibility of resolving conflicts is left to the programmer.

3.3 Specification

We now present the specification of *Spray* programming using the Abstract State Machine formalism [Börger and Stärk, 2003]. We chose it as formalism to describe the meaning of the operations because it comes with an intrinsic notion of state, which is essential for our treatment since an object has a

[4] As reported in [Brewer, 2012], some systems delay risky operations during partitioning.

related state[5]. We are not strictly interested in the final result of a computation but in how a state evolves. Furthermore, in ASM it is possible to express mathematical structures useful to describe data structures abstracting from implementation details.

We consider a system of processes where each process has a replica of part of the whole persistent distributed heap. Furthermore, we assume that each process has it own heap memory and local storage. The purpose of this specification is to describe how objects persisted on the local storage are replicated on the heap memory, that can be understood as a working memory.

Each ASM rule is introduced together with its description in natural language. We use framed boxes as graphic notation to specify the set to which an ASM rule belongs:

Dotted frames are used for **internal** rules, which define the internal parts of *Spray* and are not exposed;

Dashed frames are used for **extension** rules, which extend some parts of the language to augment with *Spray*;

Solid frames are used for **interface** rules, which define the interface exposed to the programmer.

We denote domains by words beginning with a capital letter, i.e., *Dom*. Elements of a domain are denoted by lowercase words, i.e., $elem \in Dom$. Functions are denoted by words in italics and have a number of parameters depending on their arity. Rules are denoted by uppercase words. Specific value of a domain are denoted using a sans serif font.

Uid is the domain of unique identifiers. An unique identifier is the identity of an object and it is used to refer to it. *FieldNames* is the domain of field names of objects. *Boolean* is the obvious domain of boolean value, with *true* and *false* as elements.

We have two functions to represent the heap memory and the local storage: *heap* and *local*, respectively. Furthermore, we have an enumeration of the possible operations, denoted in verbatim.

The creation of a new object yields a new identifier that is the object's identity; allocates, but does not initialise, the memory for its fields and return the identifier. After that, the fields of the object can be initialised.

```
CreateObject(fields) =
  let o = new (UId) in
    fields(o) := fields
    result := o
```

[5]This state is represented by the values of its fields.

Scopes are a special kind of objects and are handled differently. The creation of a new scope requires to specify in which scopes persist the new one. A scope is an object, and therefore has its identity, with an environment and a memory, represented respectively by $\mathsf{env} \in \mathit{FieldNames}$ and $\mathsf{mem} \in \mathit{FieldNames}$. Besides this, it is bound up with the underlying local storage, which is organised in scopes. For that reason, after the scope creation, the new scope must be stored on the local storage and persisted on all the scopes of the set passed as parameter. From these scopes it will be possible to retrieve the newly created scope later.

The STORE and PERSIST rules are defined later in this section. The former rule performs an operation on the local storage, while the latter persists an object into a scope. We assume that the set of scopes passed as parameter is not empty, otherwise an error occurs. How to deal with this error situation it is left to the implementation.

```
CreateScope(scopes) =
  let scp = CreateObject({env, mem}) in
    Store(createScope, scp)
    forall s in scopes do
      Persist(scp, s)
    result := scp
```

Scopes created with the previous rule are the so called "single" scopes. On the other hand, *allOf* scopes are not explicitly created; they are derived from the set of open scopes at a certain time.

It must be possible for the programmer to express an *allOf* scope. This is the purpose of the following functions, which returns the unique identifier of the *allOf* scope defined by the set of scopes passed as parameter. If the specified set is a singleton, and so has only one element, this function returns the identifiers of that scope. In a similar way, if the identifier of an *allOf* scope is in the set, it is replaced by the identifiers of the scopes defining it[6].

```
allOf(scopes)
```

The set of single open scopes at each time is given by the set of scopes that are reachable from the local *root* scope and are open. The set of *allOf* open scopes are implicitly defined and so not included (they are all the possible combinations of two or more scopes in the open ones). We assume the existence of the internal *ScopesFrom* function which returns all the scopes reachable from a given scope.

[6]This precaution is needed to limit the possible combinations and to not have *allOf* scopes defined by other *allOf* scopes.

$$openScopes = \{s \in ScopesFrom(root) \mid isOpen(s)\}$$

Internally, we have a function that returns the set of all open scopes: both single and *allOf*.

$$allOpenScopes = openScopes \cup \{allOf(set) \mid set \in \mathcal{P}(OpenScopes)\}$$

The following function returns the scope containing the object passed as argument, where it was persisted before or from where it was loaded in the heap. This function is not defined for transient objects, persistent objects that are not loaded in the heap and scopes.

$$scope(obj)$$

Spray requires to extend some procedures of the language to augment. The first one to consider is the procedure to get the value of a field of a specified object. If that object is not persistent (it has not been persisted in any scope before), then GetField returns the value of the object's field. On the other hand, if the object is persistent, the scope containing it must be open to return the value of the field.

In both cases, we have to manage properly the situation in which the value is an unique identifier for a persistent object that is not loaded (and so it has no associated scope). If there is an open scope that contains it, the procedure loads it and returns the identifier. In all the other cases (the value is a primitive value, the value is an identifier for a transient object or for a persistent object already loaded), the procedure simply returns the value.

We assume that *obj* is a generic object and not a scope, for which related procedures are reported in the following. In addition, for this rule there are two conditions that must be handled in the implementation. The first one is when the scope is closed, and so the object is unreachable, while the other one is when there is no open scope that contains it.

```
GetField(obj, field) =
  if isPersistent(obj) then
    if scope(obj) ≠ undef and isOpen(scope(obj)) then
      let v = heap(obj, field) in
        if isPersistent(v) and scope(v) = undef then
          if ∃s ∈ allOpenScopes : v ∈ objectOf(s) then
            Load(v)
            result := v
        else
          result := v
```

```
    else
      let v = heap(obj, field) in
        if isPersistent(v) and scope(v) = undef then
          if ∃s ∈ allOpenScopes : v ∈ objectOf(s) then
            Load(v)
            result := v
        else
          result := v
```

In a similar way, also the language procedure to set a field must be extended. Also in this case, if the object is transient, SetField just updates the location of the object's field. Instead, if the object is persistent, the scope containing it must be open. In this case, the field in the heap memory is update with the value as well as in the underlying storage for all the open scopes that contains the specified object. After that, all the callbacks registered for a change event of the object or the particular field are invoked.

As for the GetField procedure, we assume that *obj* is a generic object. For this rule, there is the case when the associated scope is closed that must be properly handled: the procedure has not effect but the implementation can decide to manage this situation as an error. Moreover, we want to underline that this procedure does not return anything. If in the language chosen to be extended with *Spray* the assignment returns a value, this rule can be extended in order to return that value.

The INVOKE rule invokes the callbacks associated with an object or a particular field of an object.

```
SetField(obj, field, val) =
  if isPersistent(obj) then
    if scope(obj) ≠ undef and isOpen(scope(obj)) then
      let old = heap(obj, field) then
        heap(obj, field) := val
        forall s in allOpenScopes with obj ∈ Objects(s) do
          Store(updateField, s, obj, field, val)
        seq
        if callbacks(change, obj) ≠ undef then
          forall c in callbacks(change, obj) do
            Invoke(c, obj, field, old, heap(obj, field))
        if callbacks(change, obj, field) ≠ undef then
          forall c in callbacks(change, obj, field) do
            Invoke(c, obj, field, old, heap(obj, field))
  else
    heap(obj, field) := val
```

The procedure that loads an object from a scope in the local storage to the heap copies the value of its fields from the scope to the memory and sets the specified scope as the one associated with the object. Each value stored must be resolved in the current environment defined by all the open scopes. This is made by the function *resolveValue*, that takes a value and the set of open scopes and return another value. If the input value is a primitive value, it is returned. Otherwise, the stored value must be resolved in the current environment and an identifier for an object is returned. After having loaded the object in the heap, all the callbacks registered for the load event of the object.

```
LoadFrom(obj, scp) =
  scope(obj) := s
  forall f in fields(obj) do
    let x = storage(scp, mem, obj, field) in
      let v = resolveValue(x, allOpenScopes) in
        heap(obj, field) := v
  seq
  forall cb in callbacks(load, obj) do
    Invoke(cb, obj)
```

In *Spray* the loading of objects from the local storage to the heap is lazy. An object is loaded only when it is required by the programmer with particular procedures, like GetField. In some cases, as in GetField, the scope from which the object must be loaded is not specified, so it is chosen among the open ones. Load chooses in a non-deterministic way a scope from the open ones that contains the object whose identity is passed as a parameter. Then it load the object in the heap from the chosen scope.

```
Load(obj) =
  choose s in allOpenScopes with obj ∈ objectsOf(s) do
    LoadFrom(obj, s)
```

A scope is open if its environment and memory are defined in the heap, and therefore they are accessible. In other words, they have been previously loaded from the storage to the heap by the Open procedure.

```
isOpen(scp) ≡
  heap(scp, env) ≠ undef and  heap(scp, mem) ≠ undef
```

Opening a scope means to load its environment and memory from the local storage to the heap. This operation has no effect if the scope is already

open or it is an *allOf* scope. In the latter case, the operation does nothing because the opening of this kind of scopes is implicit and is performed when a single scope is opened.

OPEN copies the environment and the memory of the specified scope from the local storage to the heap and notifies the underlying storage. The same operations are performed for all the *allOf* scopes that are implicitly opened, which are defined by the union of a subset of the open scopes with the one passed as argument.

This predicate returns true if the scope is actually opened or it is already open, false otherwise.

```
OPEN(scp) =
  if not isOpen(scp) and not isAllOf(scp) then
    heap(scp, env) := storage(scp, env)
    heap(scp, mem) := storage(scp, mem)
    STORE(getNotified, scp)
    forall set in P(OPENSCOPES) with set ≠ {} do
      let a = allOf({scp} ∪ set) in
        heap(a, env) := storage(a, env)
        heap(a, mem) := storage(a, mem)
        STORE(getNotified, a)
    result := true
  else
    if isOpen(scp) then
      result := true
    else
      result := false
```

Closing a scope is the opposite of opening it: its environment and memory are removed form the heap and no more accessible. As for the open rule, it is not possible to close an *allOf* scope, since that operation is implicit and it is performed when one of the single scopes that define it is closed.

To actually close a scope, the close operation has to be performed as many times as the open operation has been performed. The function *isClosable* guarantees this conditions. If a scope can be closed, the rule unloads its objects, environment and memory. These operations are performed as well as for all the *allOf* scopes which have the specified scope as member of their defining set.

This rule returns true if the scope is actually closed or it is already closed, false otherwise.

```
CLOSE(scp) =
  if isClosable(scp) and not isAllOf(scp) then
```

```
    forall set in P(OpenScopes) with scp ∈ set do
      let a = allOf(set) in
        UnloadFrom(scp)
        heap(a, env) := undef
        heap(a, mem) := undef
    UnloadFrom(scp)
    heap(scp, env) := undef
    heap(scp, mem) := undef
    result := true
  else
    if not isOpen(scp) then
      result := true
    else
      result := false
```

The UnloadFrom rule sets to undefined the *scope* function for all the objects loaded from the specified scope and unloads these objects.

```
UnloadFrom(scp) =
  forall o in objectOf(scp) with scope(o) = scp do
    scope(o) := undef
    forall f in fields(o) do
      heap(o, f) := undef
```

Internally, we have two procedures to handle the persistence of an object inside a scope. Lock ensures that an object will not be removed from a scope, whereas Unlock discharges this guarantee and so it will eventually be removed.

```
Lock(obj, scp)
Unlock(obj, scp)
```

The purpose of Persist is to make a transient object persistent by saving its state in a scope on the underlying storage. This rule is also used to persist in the specified scope an object already persisted in another scope. This operation has no effect if the object passed as parameter has already been persisted in the scope passed as argument.

The rule first locks the object in the specified scope and then notifies the local storage of the operation, distinguishing between scope objects and generic objects because the are stored differently[7]. In addition, if the specified scope is open, it adds the object to the scope's memory, set it as the

[7] An object may be persisted in different scopes and in each of them there is a possible

containing scope (only if it is not already set[8]) and invokes the callbacks associated with this operation, based on the type of the object.

The case when the object passed as parameter is persistent but is not loaded in the heap, and so it has no associated scope, should be managed properly. The problem is that the object has no state and so it may not be persisted. How to deal with this special case is left to the implementation.

```
PERSIST(obj, scp) =
  if obj ∉ storage(scp, mem) and
  (not isPersistent(obj) or scope(obj) ≠ undef) then
    LOCK(obj, scp)
    if isScope(obj) then
      STORE(addScope, scp, obj)
    else
      STORE(addObject, scp, obj)
    if isOpen(scp) then
      heap(scp, mem) := heap(scp, mem) ∪ {obj}
      if scope(obj) = undef then
        scope(obj) := scp
      seq
      if isScope(obj) and callbacks(addScope, scp) ≠ undef then
        forall c in callbacks(addScope, scp) do
          INVOKE(c, scp, obj)
      if not isScope(obj) and callbacks(addObject, scp) ≠ undef
      then
        forall c in callbacks(addObject, scp) do
          INVOKE(c, scp, obj)
```

An object that has been persisted in a scope may be released later. This is performed by the RELEASE rule. It is possible to release an object from a specified scope only if the object is in the scope's memory and it is releasable. The function *isReleasable* returns true only if no other processes refer to the object passed as parameter. If these conditions are matched, the object is unloaded from the heap, if its associated scope is the one specified, and is unlocked from that scope. It will eventually be deleted from it in the storage.

```
RELEASE(obj, scp) =
  if obj ∈ storage(scp, mem) and isReleasable(obj, scp) then
    if scope(obj) = scp then
```

different version of its state, whereas a scope is invariant with respect to the scopes where it is persisted.

[8]This is the case of a transient object that has never been persisted before.

```
    scope(obj) := undef
    forall f in fields(o) do
      heap(o, f) := undef
UNLOCK(obj, scp)
```

The following rules regard scopes. Before describing them, we explain why there are different versions of some of them with a different semantics. The first reason is that there are two type of scope: single and *allOf*. *Spray* provides methods whose semantics regard both a single scope and all the *allOf* scopes that have the single scope as member of theirs defining set. The rules with this semantics have ALL as prefix.

The other reason is that some operations load the objects of the scope in the heap, but some of them may have been already loaded from other scopes and therefore another state of those objects is already in the heap. *Spray* leaves the decision to reload from another scope the state of an already loaded object or keep the current state in the heap to the programmer. In other words, there is a version of some operation that loads an object only if it is not already loaded and another one that loads the object in any case. This latter version shadows the already loaded state of the object, which in any case is persisted in other scopes and may be made available again if the last scope is closed[9]. The rules where the loading is performed anyway have MASK as suffix in order to point out that already loaded objects will be masked.

NAMES returns the set of names defined in the environment of the scope passed as argument, if it is open[10]. The empty set is returned otherwise.

```
NAMES(scp) =
  if isOpen(scp) then
    result := keys(heap(scp, env))
  else
    result:= {}
```

ALLNAMES returns the set of names defined in the environment of the scope passed as argument, if it is open and an *allOf* scope. However, if the scope is open and it is a single scope, this rule returns the union of the set of names of the scope's environment with all the set of names of the environments of *allOf* scopes which have the specified single scope as member of theirs defining set. If the scope is close, the rule returns the empty set.

[9] This is similar to what happens in variable shadowing when a name within a certain scope shadows a name in an outer scope.

[10] It may be that different names are bound to the same object.

```
AllNames(scp) =
  if isOpen(scp) then
    if isAllOf(scp) then
      result := keys(heap(scp, env))
    else
      result := keys(heap(scp, env)) ∪
      {k ∈ keys(heap(allOf(set), env)) | set ∈ P(OpenScopes) ∧
      scp ∈ set}
    else
      result:= {}
```

A name may be bound to an object in a scope using the Bind rule. To bind a name, the scope must be open and must contain the object passed as parameter. If these conditions are matched, the rule binds the name to the object in the scope's environment and stores this binding on the local storage, distinguishing the creation of a new binding from the update of an existing one. After that, the callbacks registered for the event that has occurred on the specific scope are invoked, distinguishing between new binding and changed ones.

```
Bind(name, obj, scp) =
  if isOpen(scp) and obj ∈ heap(scp, mem) then
    let old = heap(scp, env)[name] in
      heap(scp, env)[name] := obj
      if name ∉ keys(heap(scp, env)) then
        Store(addBind, scp, name, obj)
        seq
        if callbacks(addName, scp) ≠ undef then
          forall c ∈ callbacks(addName, scp) do
            Invoke(cb, scp, name, undef, obj)
      else
        Store(updateBind, scp, name, obj)
        seq
        if callbacks(updateName, scp) ≠ undef then
          forall c ∈ callbacks(updateName, scp) do
            Invoke(cb, scp, name, old, obj)
```

In some cases the programmer wants only to define a name in a scope's environment, without binding it to a specific object. For this reason, in *Spray* there is a particular object, with its unique identifier, which has not state or behaviour: the NULL object.

NULL $\in$ *Uid*

The Resolve rule resolves a name in the environment of the scope passed as parameter. In order to resolve the name, the scope must be open and its environment must contain that name. If the name is bound to an object, this object is returned. Furthermore, if this object is not already loaded in the heap, it is loaded from the specified scope. However, if it is already loaded (its associated scope is defined), the object returned has a state that may differ from the one stored in the scope passed as parameter.

This rule avoids possible problems that can arise from the replacement of the state of an object in the heap. The decision of how to deal with the cases in which the scope is closed or its environment has not the required name is left to the implementation.

```
Resolve(name, scp) =
  if isOpen(scp) then
    if name ∈ keys(heap(scp, env)) then
      let o = heap(scp, env)[name] in
        if not isScope(o) and scope(o) = undef then
          LoadFrom(o, scp)
        result := o
```

The ResolveMask rule is similar to the previous one, but it loads the object in any case.

```
ResolveMask(name, scp) =
  if isOpen(scp) then
    if name ∈ keys(heap(scp, env)) then
      let o = heap(scp, env)[name] in
        if not isScope(o) then
          LoadFrom(o, scp)
        result := o
```

The AllResolve rule resolves a name in the environment of the scope passed as parameter. However, if the name is not defined in that environment (and therefore it can not be resolved locally to the scope) and the scope is a single scope, the name is resolved in one of the environments of the *allOf* scopes whose defining set has the single scope as member. The choice of the scope in which to resolve the name is made in a non-deterministic way among the ones whose environment has that name. The object bound to the name is loaded if it has not been previously loaded from another scope.

```
AllResolve(name, scp) =
  if isOpen(scp) then
    if name ∈ keys(heap(scp, env)) then
      let o = heap(scp, env)[name] in
        if not isScope(o) and scope(o) = undef then
          LoadFrom(o, scp)
        result := o
    if name ∉ keys(heap(scp, env)) and not isAllOf(scp) then
      let scopes = {allOf(set) | set ∈ P(OpenScopes) ∧ scp ∈
      set} in
        if ∃s ∈ scopes : name ∈ keys(heap(s, env)) then
          choose a in scopes with name ∈ keys(heap(a, env)) do
            let o = heap(a, env)[name] in
              if not isScope(o) and scope(o) = undef then
                LoadFrom(o, a)
              result := o
```

The following rule is similar to the previous one, but it loads the object in any case.

```
AllResolveMask(name, scp) =
  if isOpen(scp) then
    if name ∈ keys(heap(scp, env)) then
      let o = heap(scp, env)[name] in
        if not isScope(o) then
          LoadFrom(o, scp)
        result := o
    if name ∉ keys(heap(scp, env)) and not isAllOf(scp) then
      let scopes = {allOf(set) | set ∈ P(OpenScopes) ∧ scp ∈
      set} in
        if ∃s ∈ scopes : name ∈ keys(heap(s, env)) then
          choose a in scopes with name ∈ keys(heap(a, env)) do
            let o = heap(a, env)[name] in
              if not isScope(o) then
                LoadFrom(o, a)
              result := o
```

The internal function *objectOf* return the set of generic objects (not scope) that a scope contains.

$$objectsOf(scp) = \{o \in heap(scp, \mathsf{mem}) \mid \textbf{not } isScope(o)\}$$

The following rule returns the set of generic object (not scope) that are contained in the scope passed as parameter, if it is open. This rule also loads the objects from the scope that are not already loaded in the heap. An empty set is returned if the scope is closed.

```
Objects(scp) =
  if isOpen(scp) then
    let objs = objectsOf(scp) in
      forall o in objs do
        if scope(o) = undef then
          LoadFrom(o, scp)
      result := objs
  else
    result := {}
```

The ObjectsMask rule returns the set of generic object of the specified scope, if it is open. However, it loads all the objects from the scope to the heap.

```
ObjectsMask(scp) =
  if isOpen(scp) then
    let objs = objectsOf(scp) in
      forall o in objs do
        if scope(o) ≠ scp then
          LoadFrom(o, scp)
      result := objs
  else
    result := {}
```

AllObjects returns the set of generic object of the specified scope, if it is open and an *allOf* scope. However, if the scope is open and it is a single scope, this rule returns the union of the set of objects of the specified scope with all the set of generic objects of *allOf* scopes whose defining set has the single scope as member. Objects not already loaded in the heap are loaded by this rule from the scope that contains them. As for the previous rules, if the scope is closed, an empty set is returned.

```
AllObjects(scp) =
  if isOpen(scp) then
    if isAllOf(scp) then
      let objs = objectsOf(scp) in
        forall o in objs do
          if scope(o) = undef then
```

```
            LoadFrom(o, scp)
        result := objs
    else
      let objs = objectsOf(scp) ∪ {o ∈ objectsOf(allOf(set)) | set ∈
      P(OpenScopes) ∧ scp ∈ set} in
        forall o in objs do
          if scope(o) = undef then
            LoadFrom(o, scp)
        result := objs
  else
    result := {}
```

The AllObjectsMask rule is similar to the previous one, but it loads all the objects from the scope to the heap.

```
AllObjectsMask(scp) =
  if isOpen(scp) then
    if isAllOf(scp) then
      let objs = objectsOf(scp) in
        forall o in objs do
          if scope(o) ≠ scp then
            LoadFrom(o, scp)
        result := objs
    else
      let objs = objectsOf(scp) ∪ {o ∈ objectsOf(allOf(set)) | set ∈
      P(OpenScopes) ∧ scp ∈ set} in
        forall o in objs do
          if scope(o) ≠ scp then
            LoadFrom(o, scp)
        result := objs
  else
    result := {}
```

The *Contains* predicate returns true if the specified scope is open and it contains the object passed as parameter. False is returned otherwise.

$$Contains(obj, scp) \equiv isOpen(scp) \textbf{ and } obj \in heap(scp, \mathsf{mem})$$

The purpose of the Fetch rule is to load the state of the specified object from the scope passed as parameter. If the object is not a scope and it is contained in the indicated scope, it is loaded from it. In the case of a scope object the rule just returns it.

The decision on how to deal with the case of a not contained object is left to the implementation.

```
FETCH(obj, scp) =
  if isOpen(scp) and obj ∈ heap(scp, mem) then
    if not isScope(obj) and scope(obj) ≠ scp then
      LOADFROM(obj, scp)
    result := obj
```

The SCOPES rule returns the set of scopes that are contained in the scope passed as parameter, if it is open. The empty set is returned otherwise.

```
SCOPES(scp) =
  if isOpen(scp) then
    result := {s ∈ heap(scp, mem) | isScope(s)}
  else
    result := {}
```

ALLSCOPES returns the set of scopes contained in the scope passed as parameter, if it is open and an *allOf* scope. However, if the scope is open and it is a single scope, the rule returns the union of the set of scopes of the specified scope with all the set of scopes of the *allOf* scope whose defining set has the single scope as member. If the scope passed as parameter is closed, the empty set is returned.

```
ALLSCOPES(scp) =
  if isOpen(scp) then
    if isAllOf(scp) then
      result := {s ∈ heap(scp, mem) | isScope(s)}
    else
      result := {s ∈ heap(scp, mem) | isScope(s)} ∪
      {s ∈ heap(allOf(set), mem) | isScope(s) ∧ set ∈
      P(OPENSCOPES) ∧ scp ∈ set}
  else
    result := {}
```

Spray adopts a publish-subscribe pattern to notify the programmer that a persistent object (scope or not) has changed in the heap.

The are two version of the SUBSCRIBE rule. The former takes an event, where *event* ∈ *PersistentEvent*, an object and a function to call if the event on that particular object occurs. The latter, in addition, takes a field that limits the calling of the function to the events that occur on the specified field of the object. In any case, the function is added to the set of callbacks only

if the object passed as parameter is a scope or a generic persistent object. For the latter version, it is also required that the field specified is one of the object's fields.

SUBSCRIBE($event, obj, cb$) =
  if $isScope(obj)$ **or** $isPersistent(obj)$ **then**
    $callbacks(event, obj) := callbacks(event, obj) \cup \{cb\}$

SUBSCRIBE($event, obj, field, cb$) =
  if $isScope(obj)$ **or** $isPersistent(obj)$ **then**
    if $f \in fields(obj)$ **then**
      $callbacks(event, obj, field) := callbacks(event, obj, field) \cup \{cb\}$

The UNSUBSCRIBE and ALLUNSUBSCRIBE rules are used to unsubscribe functions related to the occurrence of an event on a particular object. As for the previous rules, there are two versions of each: event of the specified object and its restriction to a single field. Also in these rules, it is required that the object passed as parameter is a scope or a generic persistent object.

UNSUBSCRIBE($event, obj, cb$) =
  if $isScope(obj)$ **or** $isPersistent(obj)$ **then**
    if $cb \in callbacks(event, obj)$ **then**
      $callbacks(event, obj) := callbacks(s, obj) \setminus \{cb\}$

UNSUBSCRIBE($event, obj, field, cb$) =
  if $isScope(obj)$ **or** $isPersistent(obj)$ **then**
    if $field \in fields(obj)$ **and** $cb \in callbacks(event, obj, field)$ **then**
      $callbacks(event, obj, field) := callbacks(event, obj, field) \setminus \{cb\}$

UNSUBSCRIBEALL($event, obj$) =
  if $isScope(obj)$ **or** $isPersistent(obj)$ **then**
    $callbacks(event, obj, field) := \{\}$

UNSUBSCRIBEALL($event, obj, field$) =
  if $isScope(obj)$ **or** $isPersistent(obj)$ **then**
    if $field \in fields(obj)$ **then**
      $callbacks(event, obj, field) := \{\}$

The STORE internal rule is used to store on the local storage scopes and objects, making them persistent. It also sends local changes to the persistent distributed storage. The particular actions to perform depends on the operation to execute, but in general they consist in updating the local

storage and transmitting the changes to the remote one.

Instead of explaining each action to perform for a given operation, we describe only the actions that are different from the one performed on the heap and put aside calls of the SEND rule. This rule takes a type of operation and its parameters, and transmits them to the persistent remote storage.

The creation of a scope in the local storage initialises its environment to the empty map and its memory to the empty set. The operation of adding a scope to another one stores the scope into the memory of the specified one. Instead, adding an object to a scope requires to store both the object and its current state into the memory of the specified scope.

The updating of an object's field in the local storage concerns the particular state of that object in the specified scope. The value previously stored for the indicated field is combined with the new one by the *update* function. This function returns the new value, if at least one of them is a primitive value. However, if the new value is a unique identifier and the old one represent a value to resolve in the environment defined by a set of open scopes, this function combines them[11].

Finally, a `getNotified` operation does not change the local storage, but sends the request to the persistent remote storage to be notified of changes to objects contained in the specified scope.

```
STORE(op, data) =
  if op = createScope then
    let scp = data in
      storage(scp, env) := {→}
      storage(scp, mem) := {}
      SEND(createScope, scp)
  if op = addBind then
    let scp, name, obj = data in
      storage(scp, env)[name] := obj
      SEND(addBind, scp, name, obj)
  if op = updateBind then
    let scp, name, obj = data in
      storage(scp, env)[name] := obj
      SEND(updateBind, scp, name, obj)
  if op = addScope then
    let scp, s = data in
      storage(scp, mem) := storage(scp, mem) ∪ {s}
      SEND(addScope, scp, s)
  if op = addObject then
    let scp, obj = data in
```

[11] In some sense, *update* and *resolveValue* are opposite function. The former combines identifiers, whereas the latter resolve that composed value to return a single identifier.

```
        storage(scp, mem) := storage(scp, mem) ∪ {obj}
        forall f in fields(obj) then
          storage(scp, mem, obj, f) := heap(obj, f)
        SEND(addObject, scp, obj, pack(obj))
    if op = updateField then
      let scp, obj, field, old, val = data in
        let old = storage(scp, mem, obj, field) in
          storage(scp, mem, obj, field) := update(old, val)
          SEND(updateField, scp, obj, field, val)
    if op = getNotified then
      let scp = data in
        SEND(getNotified, scp)
```

Besides the previous rule, there is the RECEIVE rule that applies locally operations received from the persistent remote storage. This rule updates the local storage and, if the scope associated with the changed object is open, also updates the heap memory and invokes all the callbacks registered for that particular event. We assume a queue of messages from which this rule removes the first one, calling the DEQUEUE rule. If the queue is empty, the *hasMessages* location return false and the rule does not do nothing, until a message arrives. When this happens, the specific operation is handled in the proper way.

Except for `setScope` and `setRoot`, the other operations are the same handled by the STORE rule and perform the same action on the local storage. Moreover, if the scope affected by the change is open or the scope that contains the changed object is open, that change is also applied to the heap and the registered callbacks for that specific event are invoked.

The `addObject` operation is the only operation, among the already discussed ones, that requires a further explanation. After having stored the object into the local storage, if the scope to which the object has been added is open, the callbacks related to this event are invoked. However, two cases must be distinguished: if a version of the object is already loaded or not. In this second case, the object is not loaded in the heap and the callbacks for the *addObject* event are invoked. The programmer may decide to load it using the FETCH operation. Also in the other case the received state of the object is not loaded but the callbacks related to the *addObjectLoaded* event are invoked instead. The loaded version of the object is passed as parameter to these callbacks as well as the scope to which it has been added. The decision to load the state of the object from that scope is left to the programmer.

Finally, the `storeScope` and `storeRoot` operations are similar because both store into the local storage the environment and the memory of the received scope. However, the latter is special because it concerns the root

scope and has to update the location containing it, from which it can be later retrieve.

```
RECEIVE =
  if hasMessages then
    let op, data = DEQUEUE(Messages) in
      if op = addBind then
        let scp, name, obj = data in
          storage(scp, env)[name] := obj
          if isOpen(scp) then
            heap(scp, env)[name] := obj
            if callbacks(addBind, scp) ≠ undef then
              forall c ∈ callbacks(addBind, scp) do
                INVOKE(cb, scp, name, obj)
      if op = updateBind then
        let scp, name, obj = data in
          storage(scp, env)[name] := obj
          if isOpen(scp) then
            let old = heap(scp, env)[name] in
              heap(scp, env)[name] := obj
              if callbacks(updateBind, scp) ≠ undef then
                forall c ∈ callbacks(updateBind, scp) do
                  INVOKE(cb, scp, name, old, obj)
      if op = addScope then
        let scp, s = data in
          storage(scp, mem) := storage(scp, mem) ∪ {s}
          if isOpen(scp) then
            heap(scp, mem) := heap(scp, mem) ∪ {s}
            if callbacks(addScope, scp) ≠ undef then
              forall c in callbacks(addScope, scp) do
                INVOKE(c, scp, s)
      if op = addObject then
        let scp, o, values = data in
          storage(scp, mem) := storage(scp, mem) ∪ {o}
          fields(o) := keys(values)
          forall f in fields(o) do
            storage(scp, mem, o, f) := values[f]
          if isOpen(scp) then
            if scope(o) = undef and
            callbacks(addObject, scp) ≠ undef then
              forall c in callbacks(addObject, scp) do
                INVOKE(c, scp, o)
            if scope(o) ≠ undef and
```

```
            callbacks(addObjectLoaded, scp) ≠ undef then
              forall c in callbacks(addObjectLoaded, scp) do
                INVOKE(c, scp, o)
      if op = updateField then
        let scp, obj, field, val = data in
          let oldstored = storage(scp, mem, obj, field) in
            storage(scp, mem, obj, field) := update(oldstored, val)
            if scope(obj) ≠ undef and scope(obj) = scp and
            isOpen(scp) then
              let {new = storage(scp, mem, obj, field),
              old = heap(obj, field)} in
                let v = resolveValue(new, AllOpenScopes) in
                  heap(obj, field) := v
                  if callbacks(change, obj, field) ≠ undef then
                    forall c in callbacks(change, obj, field) do
                      INVOKE(c, obj, field, old, v)
                  if callbacks(change, obj) ≠ undef then
                    forall c in callbacks(change, obj) do
                      INVOKE(c, obj, field, old, v)
      if op = storeScope then
        let scp, renv, rmem = data in
          storage(scp, env) := renv
          storage(scp, mem) := rmem
      if op = storeRoot then
        let r, renv, rmem = data in
          root := r
          storage(r, env) := renv
          storage(r, mem) := rmem
```

The INIT rule must be used to initialise *Spray*, and therefore it must be called before any other rules that regards the persistent distributed heap. This rule sends a request for the root specifying which scopes include, namely the one of the application, the one of the device and the one of the user. Then, it waits that the root is received to open it.

```
INIT(appId, deviceId, userId) =
  let {app = getScopeId(appId),
  device = getScopeId(deviceId),
  user = getScopeId(userId)} in
    SEND(getRoot, {app, device, user})
    if root ≠ undef then
      OPEN(root)
```

The programmer can retrieve the root with the following rule.

ROOT = **result** := *root*

The PERSISTENT rule manages the persistent storage and transmits the updates among the replicas. We assume that the root scope exists and there is a function (*getRoot*) that returns it. We also assume that received messages are stored in a queue. If there are received messages, the first one is removed from the queue, calling the DEQUEUE rule, and the related operation is performed. In this case we suppose that DEQUEUE also returns the sender of the message.

This rule uses two functions to store which replicas have a certain scope and which scopes are stored in a particular replica, these are *replicas* and *scopes* respectively. These functions are updated in the operations that concerns scopes: explicitly in `createScope` and implicitly `addScope`, `getRoot` and `getNotified`.

The `createScope` operation is the only one that transmits no messages to the replicas. It creates an empty scope in the storage and initialises the replicas function for that scope to the singleton set where the element is the sender of the operation. It also adds the scope to the set of scopes replicated in the sender. For the `addBind`, `updateBind` and `addObject` operations, the change is applied to the local storage and sent to all the replicas where the specified scope is replicated. In the same manner, an `updateField` operation is applied to the specified object of a certain scope into the storage and send to all the replicas having that scope. Here and in the following, an operation is not transmitted to the replicas that had originally forwarded it.

An `addScope` operation requires to add a scope to the specified scope's memory into the storage, to send the operation to all the other replicas and to send, using the SENDSCOPE rule, the added scope to all the replicas that have not replicated it yet.

The `getNotified` operation specified that the sender opened the scope passed as parameter. As a result, all the scopes contained in that scope must be sent to it as well as all the *allOf* scopes it may open from this moment on. First all the scopes contained in the specified one, that are not already replicated in the sender, are transmitted to it. Then, the other *allOf* scopes which may be opened due to the updated set of single scopes are transmitted to the sender.

Finally, the `getRoot` operation handles the request for the root made by a replica. Although the set of scopes replicated on the sender is initialised to the singleton set containing the root, the sender is not added to the replicas of the root. The reason why this is not done is that objects (scope or not) added to the root would be transmitted to all the replicas. Instead, we want that a replica's root has only the scopes specified in the request. The root is transmitted to the sender with a restricted environment and memory, using

the functions *restrictEnv* and *restrictMem*. These functions respectively that take an environment and a memory and restrict it based on the set of scopes passed as argument. These scopes are then transmitted as well as all the possible *allOf* scopes given by the union of these ones with the root.

```
PERSISTENT =
  let root = getRoot() in
    if hasMessages then
      let op, data, sender = DEQUEUE(Messages) then
        if op = createScope then
          let scp = data in
            storage(scp, env) := {}
            storage(scp, mem) := {}
            replicas(scp) := {sender}
            scopes(sender) := scopes(sender) ∪ {scp}
        if op = addBind then
          let scp, name, obj = data in
            storage(scp, env)[name] := obj
            forall r in replicas(scp) \ {sender} do
              SENDTOREPLICA(r, addBind, scp, name, obj)
        if op = updateBind then
          let scp, name, obj = data in
            storage(scp, env)[name] := obj
            forall r in replicas(scp) \ {sender} do
              SENDTOREPLICA(r, updateBind, scp, name, obj)
        if op = addScope then
          let scp, s = data in
            storage(scp, mem) := storage(scp, mem) ∪ {s}
            forall r in replicas(scp) with s ∉ scopes(r) do
              SENDSCOPE(s, r)
            forall r in replicas(scp) \ {sender} do
              SENDTOREPLICA(r, addScope, scp, s)
        if op = addObject then
          let scp, o, values = data in
            storage(scp, mem) := storage(scp, mem) ∪ {o}
            fields(o) := keys(values)
            forall f in fields(o) do
              storage(scp, mem, o, f) := values[f]
            forall r in replicas(scp) \ {sender} do
              SENDTOREPLICA(r, addObject, scp, o, values)
        if op = updateField then
          let scp, obj, field, val = data in
            let old = storage(scp, mem, obj, field) in
```

```
            storage(scp, mem, obj, field) := update(old, val)
            forall r in replicas(s) \ {sender} do
              SendToReplica(r, updateField, scp, obj, field, val)
      if op = getNotified then
        let scp = data in
          forall s in storage(scp, mem) with isScope(s) and
          s ∉ scopes(sender) do
            SendScope(s, sender)
          seq
          let singles = {s ∈ scopes(sender) | not
          isAllOf(s)} in
            forall set ∈ P(singles) with set ≠ {} do
              let a = allOf(set) in
                if a ∉ scopes(sender) then
                  SendScope(s, sender)
      if op = getRoot then
        let scopes = data in
          let {renv = restrictEnv(storage(root, env), scopes),
          rmem = restrict(storageMem(root, mem), scopes)} in
            scopes(sender) := {root}
            SendToReplica(sender, storeRoot, root, renv, rmem)
            forall s in scopes do
              SendScope(s, sender)
            forall set ∈ P(scopes ∪ {root}) with set ≠ {} do
              let a = allOf(set) in
                SendScope(s, sender)
```

The SendScope rule transmits the specified scope to a replica. More in details, it adds the replicas to the set of replicas associated to a scope, adds the scope to the set of scopes of that replica, sends the scope with its environment and memory and each of its object, distinguishing between scopes and generic objects.

```
SendScope(scp, replica) =
  let {renv = storage(scp, env), rmem = storage(scp, mem)} in
    replicas(scp) := replicas(scp) ∪ {replica}
    scopes(replica) := scopes(replica) ∪ {scp}
    SendToReplica(replica, storeScope, scp, renv, rmem)
```

3.4 Some properties

In the following we state and prove some basic properties about *Spray*, starting from the trivial ones.

Property 1. *A scope is open or is closed.*

Proof. The property follows from the rules that manipulate a scopes and it is proved considering that a scope is open if its environment and its memory are loaded in the heap. After being created, the scope's environment and memory are not initialised in the heap, thus the scope is closed. Open changes the status of a scope from closed to open, whereas Close does the vice versa. The other rules does not change the status of the scope. □

Property 2. *At a given time, there is only one version of the state of an object in the heap or the object is not loaded in the heap.*

Proof. The property is proved considering the *scope* function and the rules that update it. A persistent object that is not loaded has no associated scope, otherwise the function returns the scope where the object has been persisted or from where it has been loaded. The rules that assign a scope to the function are Persist, which updates *scope* only if it is undefined for the specific object, and LoadFrom, which is performed only if the object is not loaded in the heap. The UnloadFrom rule instead sets the function to undefined. □

Property 3. *If a process executes a* SetField *rule assigning the value x to a field of a given object (which is persisted in a scope) and no other processes execute that rule for the same field of that object (in the specific scope), then a consecutive* GetField *rule performed by the process returns the value x.*

Proof. The property follows from the locality of the operation, assuming that no other processes assign another value to the specific field of the given object. This means that the Receive rule will not receive an operation for the field of that object and as a consequence the value stored in the heap of the considered process will not be modified. □

Property 4. *An* allOf *scope is open if and only if all the single scopes that define it are open.*

Proof. The property follows from the definition of the Open rule. It does not open an *allOf* scope if it is passed as arguments. Instead, it opens the *allOf* scopes defined by the union of a no-empty subset of the open scopes with the one to open. □

Property 5. *If a process executes a* SetField *rule assigning the value x to a field of a given object (which is persisted in a scope) and no other processes*

execute that rule for the same field of that object (in the specific scope), then this update will eventually arrive to other processes which have requested to be notified for the scope containing the object.

Proof. The property follows from the definition of SetField, Store and Persistent. SetField calls the Store rule specifying the operation to perform and its arguments. Store calls the Send rule passing as parameters the operation and its arguments. Send transmits the operation message to the remote persistent storage[12]. Persistent receives the message and forwards it to all the replicas (processes) associated with the scope specified in the operation. □

Property 6. *If a process executes a* SetField *rule assigning the value x to a field of a given object (which is persisted in a scope) and no other processes execute that rule for the same field of that object (in the specific scope), then a* GetField *rule performed on the same field of the object (in the specific scope) by any process eventually returns the value x.*

Proof. The property follows from the definition of Store, Persistent and receive, assuming that no other processes assign another value to the specific field of the given object. The operation transmitted by the Store rule is eventually received from the persistent remote store[12]. The Persistent rule receives the `updateField` operation and forwards it to the other replicas of the specified scope. It is worth noting that any later `storeScope` for the indicated scope will contain the update. The message eventually arrives to all the replicas. The Receive rule receives the update and then applies it to the local store. At this point, if a replicas performs a GetField rule, it returns the value of the field of the specific object (of the particular scope) that was initially transmitted. □

Property 7. *A process does not receive operations regarding scopes it can not reach and it can not open.*

Proof. This property follows from the definition of Store and Persistent. Store send a request to be notified of changes to a specific scope as consequence of the opening of that scope. Persistent responds transmitting back the scopes contained in the specified scope as well as the additional *allOf* scopes that may be open. It also registers for which scopes a replica want to be notified, and therefore it will not be notified for other scopes. □

[12]We assume that the communication is reliable and no messages are lost.

4

SprayJS

> *"To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages; . . . "*
>
> – Robert W. Floyd, *The Paradigms of Programming*

In this chapter we describe the proof-of-concept implementation of *Spray* in JavaScript, based on the specification of Section 3.3, called SprayJS.

Firstly, we provide a brief description of JavaScript and explain the reasons why this language has been chosen. Then, we outline the general characteristics of the implementation, which has been built in Node.js. Finally, we describe separately both the client-side and the server-side of SprayJS.

4.1 Remarks about JavaScript

JavaScript is an object-oriented programming language. It was originally designed to be a Web scripting language and it is widely used for client-side scripting in web browsers. It is one of the most well-known implementation of the ECMAScript language specification, a language standardized by Ecma international. Here and in the following, we refer to the version 5.1 of the standard [ECMA, 2011].

JavaScript is a dynamically typed programming language, its objects are associative arrays and has functions as first-class citizens, more precisely they are callable objects. It is a prototype-based object-oriented language and does not use classes, compared to more traditional class-based languages like Java. Inheritance is not obtained using classes, which describe behaviour, but through prototypes, that are object themselves. Every object created has an implicit reference to its prototype. Furthermore, a prototype may have non-null implicit reference to its prototype, and so on; this is called prototype chain. An object in JavaScript is a collection of properties[1] with

[1] A function that is associated with an object via a property is a method.

zero or more attributes, which determine how each property can be used. If a property is not found in the object, the prototype chain is visited until the property with the specified name is found or undefined is return otherwise.

Other than objects and functions, JavaScript provides primitive values of the following types: boolean, number, string, undefined and null. The undefined value is used when a variable has not been assigned a value, whereas the null value represents the intentional absence of any object value.

An interesting aspect of JavaScript is its flexibility. It is possible to dynamic add and remove properties to an object. This is also possible for built-in objects, if not explicitly limited through the specific attributes.

4.2 Why JavaScript?

We have chosen JavaScript as language in which implement Spray because it is a prototype-based language and is flexibile. Indeed, JavaScript allows to extend built-in objects, adding to them properties, and dynamically attach additional behaviour to properties, in a way that looks like an extension to the runtime of the language for a programmer. These characteristics had been essential to implement the core of SprayJS.

We have previously mentioned that JavaScript is most commonly used for the client-side part of web pages. However, it is also possible to use it for the server-side part. The most well-known platform that used JavaScript as language is Node.js, that has contributed to spread JavaScript also as server-side language for developing application. Node.js adopts an event-driven, non-blocking I/O model that enables the development of servers in JavaScript, without using threading[2]. As stated in [Node.js Foundation, 2015], this model is "perfect for data-intensive real-time applications that run across distributed devices". The possibility to use JavaScript both on the client-side and the server-side was a helpful consequence of our decision.

Its prototype-based nature is not a limitation but instead allows to simulate the class-based approach. Indeed, is quite common, especially for programmers with no experience in JavaScript and that are used to class-based languages, to develop an application in JavaScript adopting a class-based point of view. The basic idea is to add to a constructor's prototype, which is the first prototype of the chain, the methods to manipulate an object returned by that particular constructor[3] and to keep in the object only the state.

From a technological point of view, JavaScript is the de facto language for client-side web application. We have mentioned several time web applications as an example of application where the idea of *Spray* may be applied

[2] A Node.js process is single threaded by design, however it is possible to spawn child processes and some external modules provide native threading.

[3] A constructor is a function that are invoked in a `new` expression.

to simplify the development and take off from the programmer the explicit management of aspects non strictly related to the logic of a program. As a result, JavaScript lets us to test our idea in the scenario of web applications.

4.3 Implementation

The *Spray* specification is implemented in SprayJS as a client-server architecture, where the client-side is represented by a JavaScript library and the server-side is a Node.js server.

The library is the most important part of SprayJS because it implements the rules of the specification of Section 3.3 concerning the client-side and provides the programmer with the interface of *Spray*. The server implements the PERSISTENT rule of the specification and embodies the persistent remote storage.

In this section we describe the implementation details common to both parts of SprayJS, while in the following sections we discuss each part separately. Our main concern is about how objects are identified and transmitted between a clients and the server.

We used UUIDs, which had been briefly discussed in Section 3.2.2, as unique identifiers in the implementation. Among the different versions described in the standard, we adopt the fourth version that generates UUIDs from pseudo-random numbers. This version has been preferred to the other ones because it does not require a namespace (versions 3 and 5) and is not based on information regarding the underlying hardware (MAC address in version 1), which are not available in JavaScript. In detail, we used an implementation of the RFC 4122 [Leach et al., 2005] available both as a Node.js module and as a script for web browsers[4].

The JSON format is used as representation for transmitting the operations presented in the specification. Most of these operations have arguments that may be expressed as strings, for example the UUID of an object or the name of a binding. The type of the value of an update operation depends on the assigned value, nevertheless it is one among number, boolean, string or UUID (which is expressed by an object with a property named UUID in order to distinguish it from a generic string). In the context of the update operation, arrays are handled as if they were general objects: the value of the update is an UUID instead of the actual object.

We use the `stringify` function[5] to convert an object, representing an operation, to a JSON string to send. Indeed, `stringify` takes an object and returns a string in the JSON format representing all the sub-graph pointed by the specified object. However, we can not use this approach for all the operations to transmit. The `addObject` operation needs a more sophisticated

[4] node-uuid: `https://github.com/broofa/node-uuid`

[5] More precisely, we use `JSON.stringify`, a method of the JSON object.

handling because *Spray* adopts a shallow approach, as described in Section 3.2.3. Only the first level of the graph is needed, while `stringify` evaluates the entire graph. In order to limit the graph traversal to the first level, a "shallow" copy of the original object is passed to the `stringify` function. This shallow copy is obtained replacing any reference to an object with the UUID of that object. If the original object is an array, a shallow copy of the array is used: those elements that are objects are replaced with their UUID. After being converted to a string, we can distinguish between an object and an array because they have a different JSON representation. Furthermore, the shallow copy of the object is also stored on the local storage.

The part of the server that implements the remote persistent storage communicates with the client library using the WebSocket protocol, which provides full-duplex communication channels and is described in RFC 6455 [Fette and Melnikov, 2011]. Considering that the specification refers to communications in both direction, from a replica to the persistent storage and vice versa, this protocol has been chosen because it provides full-duplex communication. Furthermore, it facilitates the interaction between the server and a client, compared to other solutions usually based on a polling technique[6].

As far as the storage is concerned, considering that our purpose was to provide a proof-of-concept implementation and not full implementation of *Spray*, we implemented the local storage both in the library and in the server as a JavaScript object, whose properties are the persisted scopes. This object is kept in memory. A real implementation would of course use a long-term storage solution to guarantee the persistence of persistent objects.

4.4 Library

The SprayJS library implements the client-side of the architecture and exposes to the programmer the interface of the specification, let to program using the abstraction of a persistent distributed heap.

The library was developed as a Node.js module and then transformed in a script suitable for web browsers, using the Node.js module browserify[7]. This module analyses all the modules requested by the one specified as argument (performing a recursive walk of the require graph) and builds a bundle to use in web browsers. Writing the library as a module allowed to simplify the development and separate the different concerns of the implementation. Indeed, it was developed as a set of sub-modules, each of which covers a different aspect of the implementation. The main modules are:

[6] A well-known example is Ajax, where the client sends requests to the server in order to receive updates.

[7] `http://browserify.org/`

- core: defines objects and functions that are fundamental for *Spray* and exposes the subset of them that represent the interface;
- storage: provides objects representing a local storage;
- remote: provides functions to communicate with the remote persistent storage;
- spray: provides the interface of SprayJS, which is constituted by the objects and functions provided by the core module as well as other functions derived from the basic ones;
- event: provides objects to implement the publish-subscribe pattern;
- domains: provides functions to retrieve the UUID of a given name that belongs to a domain (applications, users or devices).

In the following we describe only the important details of the various modules, pointing out where they are implemented, rather than delineate each module separately.

The first issue that we had to deal with was how to combine the identity of an object in *Spray*, represented by an UUID, with its identity in the language, represented by a reference (its address in memory). Taking into account the flexibility of JavaScript and its design, we extended the Object prototype, which is the last in the prototype chain of all objects, with a function that returns the UUID of the object on which it is invoked. More precisely, the `getUUID` function returns the value of the `UUID` property of the object on which it is invoked, assigning a new generated UUID for that property, before returning, if it is not defined. Furthermore, to avoid that the programmer accidentally changes it, the writable attribute is set to false.

Listing 4.1 reports the code fragment that implements this core feature. It is executed as soon as the application code is loaded in the browser.

```
Object.defineProperty(Object.prototype, 'getUUID', {
  value: function () {
    if (this.)
      Object.defineProperty(this, 'UUID', {
        value: uuidGen(),
        enumerable: false,
        writable: false
      });
    return this.UUID;
  },
  enumerable: false
});
```

Listing 4.1 Adding UUID to JavaScript objects.

Even though not all objects at runtime have an UUID, the subset of them that are persisted have that property because they are loaded from the storage with it or they are persisted for the first time, operation that adds to them the `UUID` property.

Another issue that arises from the fact of having two identities[8] is how to retrieve the reference of an object, given its UUID. A mechanism to retrieve it is needed when a field of an object is read for the first time and the UUID stored in that field must be replaced with the reference of the associated object. To avoid copy of persistent objects already loaded, we adopt an index that has as entries the UUIDs of the already loaded persistent objects and as values a reference to them. This index is implemented as a JavaScript object in the core module.

As far as the error conditions described in the specification are concerned, except where explicitly indicates, in the implementation we have chosen to return undefined. This decision is justified by the fact of avoiding to the programmer to deal with JavaScript's exceptions.

In the core module are defined the two most important object of the library: proxy and scope. The former is used as a moniker for regular objects, while the latter represent a scope, as described in the specification.

A proxy object embodies the proxy pattern. Its responsibility is to control the access to the real object. It loads the object form a particular scope in the storage and sends to it the updates to the object's fields. However, a proxy does not point directly to the real object. In order to hide it from the programmer and to memorise from which scope the object has been loaded and what are the other open scopes that contain it, we use a middle object called "target object". The target object has a reference to the real object. Figure 4.1 shows the object diagram of a proxy at runtime.

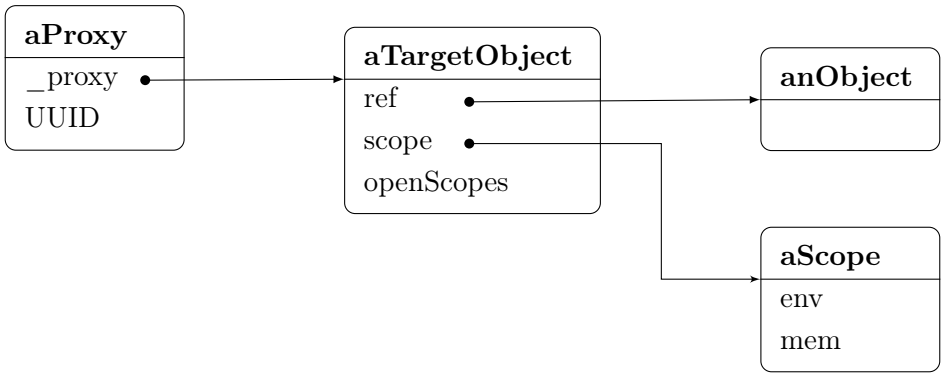

Figure 4.1: Object diagram of a proxy.

The object diagram does not report the properties of the proxy and the

[8] We may think to references as transient identities which, at runtime, univocally identifies the UUID of the object.

real objects because they depend on the actual object. The `openScopes` property of a target object is not shown in the diagram, however it points to an array containing the open scopes used as a stack.

Properties are added to a proxy object when it is created passing the target object or when (a version of) the real object is loaded from a scope that contains it. For each property of the real object, a getter-setter pair of functions is added to the proxy object. These functions implement respectively the GetField and the SetField rules of the specification in the case of a persistent (distributed) object.

In the specification *resolveValue* and *update* are unspecified. They are used respectively for resolving the value of a field in the set of open scopes and for combining the old and the new values of a field.

In our proof-of-concept implementation, we used arrays as values for fields that refers to other objects. An array contains the collection of UUIDs of the objects assigned to the field. These arrays are stored in the local storage and retrieved when an object is loaded in memory. Each field of the object to load that has an array as value is resolved in the current environment: the array is scanned backward and the proxy of the first UUID that is in one of the open scopes, and so in the index, is assigned to the field in memory. When that field is updated, the old value is replaced in the array by the new one. If no UUID is loaded, because there is no reachable object in the current environment, the field is left undefined and, when it is assigned, a new UUID is added to the end of the array. In contrast, primitive values are stored in the local storage as they are[9].

A scope object is an object with an environment and a memory. Both the properties refer to a JavaScript object that is used as associative array. The environment object has as keys the names bound to an object and as values the UUID of these objects. Instead, the memory object has as names the UUIDs of the objects contained in the scope and as values a meta object containing information regarding the actual object[10]. The state of the object, previously persisted in the scope, is stored in the storage and loaded only when needed.

In the specification we distinguished two kind of scopes: single and *allOf*. In the implementation they are constructed by two different functions, but both of them call the scope's constructor as first operation and then add another property to the (`this`) object. A single scope has as further property an array that contains a reference to the *allOf* scopes having this scope as member of their defining set. An *allOf* scope, instead, has as property an array that refers to the single scopes from which it is defined.

In order to have inheritance, to both the prototypes of single and *allOf*

[9]This means that if a primitive value is assigned to a field that contains a reference to another object, in the local storage, the associated array is replaced by the value.

[10]In particular it contains the type of the object.

scope are assigned, as prototype, the scope's prototype. Thus, properties not found on the prototype of single or *allOf* scope are looked up on the prototype of scope. In such a way, functions common to both kind of scopes are not replicated on each prototype.

In the core module, we defined three constructors: Scope, Singleton and Set. They are used respectively for creating a generic scope, a single one and a *allOf* one. As above-mentioned, the Scope function is called by both the Singleton and Set constructors. If we interpret these constructors in terms of a class-based view, they are the constructors of the relative classes. In Figure 4.2 is shown the class diagram for scopes[11], It presents how both the Singleton and the Set classes are specialisation of the more general Scope class, and the association that exists between them.

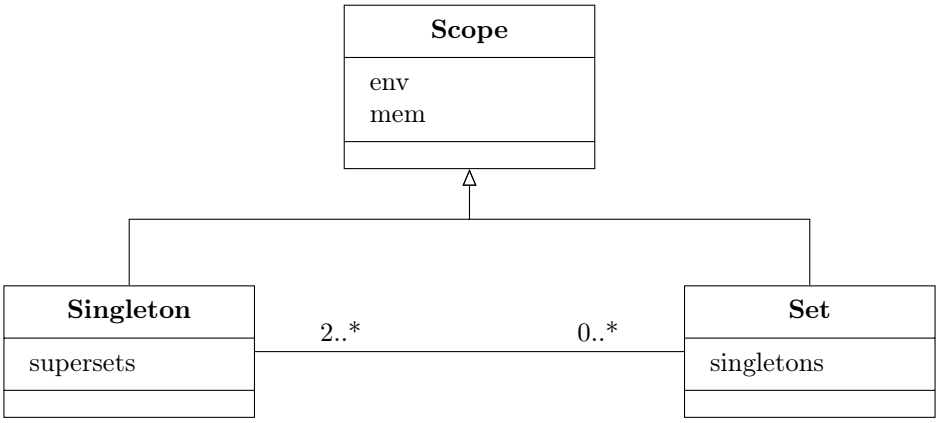

Figure 4.2: Class diagram showing inheritance between scopes.

The implementation of the operations concerning scopes follows from the specification. The only operation that is worth to mention is PERSIST because it had required a slight extension. In the specification this rule persist an object into a scope on the local storage and returns nothing; in the implementation it returns the object persisted. Returning the object is required to handle the first time that a regular object (not a scope) is persisted. The function takes the object and return its proxy. This object must be assigned to all the variables or fields that referred to the real object before the operations.

JavaScript is a liberal language and does not provide access control to the properties of an object. In order to avoid that the programmer manipulates directly a scope, the library provides a ScopeMoniker object, instead of an actual scope object. A ScopeMoniker is simply an object with one property that is the UUID of the related scope. All the functions to manipulate a ScopeMoniker are defined in the spray module and are exposed to the

[11] This is an abuse of notation, since JavaScript is prototype-based and has no classes

programmer. Internally, they retrieve the actual scope and perform the specific operation.

Scopes adopts two structures described in [Shapiro et al., 2011a] to achieve strong eventually consistency. The environment of a scope is designed as a Grow-only Set (G-Set). Names can only be added to it. However, it is possible to do a logical remove of a name binding it to the null object.

On the other hand, the memory of a scope embodies a PN-Set structure: a set where to each element is associated a counter. The first time that an object is added to the set (memory), its counter is initialised to 1. After that, further adding of the object increments the counter, while a remove decrements it. Objects counter is not strictly positive are not considered in the set. However, in order to avoid a premature delete of an object used by another execution of an application, we enforce a constraint: an instance of an application may not perform on an object more persist than release.

There are two other details that we want to report on. The first one concerns how the publish-subscribe pattern has been implemented in the event module. Node.js provides an EventEmitter object that embodies this pattern and it is used as base object for other objects provided in the API. However, an EventEmitter has only one level of events: a function is added for an event specified by a name and, when it occurs, the function is invoked.

In the specification of *Spray* we distinguished functions to call when an event occurs to an object in general and functions to call when an event occurs to a specified field of an object. In order to deal with both the cases in an unique way, we implemented an EventEmitter object that has two level of events. The first level of events is obligatory while the second one is optional. The main event is used for denoting an occurring event and the sub-event is applied to restrict the main one to a particular field. The developed EventEmitter constructor is called, as first operation, for both proxy and (generic) scope objects.

The second and last detail concerns the domains module, which contains the function used to retrieve, from the server, the UUID associated with a given name in a domain. Applications and users are identified by unique names, whereas for devices we chose to use their IP addresses. This decision has been made in order to identify univocally a device and works fine in a static assigned network, like the one where we test the implementation. It has also simplified the development. However, the problem of how identify univocally a device is orthogonal to our goal and other solutions can be applied. As an example, most operating systems provide a device unique identifier, which can be retrieved through a specific API.

4.5 Server

The SprayJS server combines a HTTP server for web resources and a WebSocket server for the persistent distributed heap. It implements the remote persistent storage and the name resolution service for three domains, namely applications, users and devices.

As described in the specification, the remote persistent storage must keep track of which replicas received operation messages should be forwarded to, i.e., the server should forward a message to replicas after having received it. The use of WebSockets, along with the assumption of online application, facilitates the management of the communication, compared to other solution for dynamic web application. As long as a WebSocket is open, the related replicas will receive operation messages. When the WebSocket is closed, the replica will be removed from the set of active replicas to which send operations.

Besides transmitting operations to all replicas, the remote persistent storage applies locally to its storage these operation and performs the additional action in order to ensure consistency.

Distinct concurrent field update operations (or name binding operations) are applied, and therefore stored in the persistent storage, as they arrive to the server. This means that only the last update (or binding) wins, whereas the other concurrent updates are lost. Adding a name to a scope's environment and adding/removing an object to a scope's memory do not require any particular action. For the former operation, if the name is not in the scope's environment, it is added to it, otherwise the operation is handled as a binding update. The latter operation adds/removes one to the counter of an object. In the implementation we assumed that each UUID generated is practically unique, and we did not manage the case of clients adding two distinct object identified by the same UUID to a scope.

5

Examples

> *"If you want to accomplish something in the world, idealism is not enough—you need to choose a method that works to achieve the goal. In other words, you need to be 'pragmatic.'"*
>
> – Richard M. Stallman, *Copyleft: Pragmatic Idealism*

In this chapter we examine common patterns for web applications and describe their implementation in SprayJS. Our purpose is to demonstrate that these patterns can be expressed in *Spray* and the burden of how objects are persisted and distributed is taken away from the programmer.

Firstly, we discuss the basic pattern for web applications that concerns the communication and the relation with the evolution of the state. Then, we take into account how the state of an application is shared, typically among users. In this context, we emphasise the value of the three specific classes of scope presented in the specification, namely application, user and device, and how they are used to express sharing in SprayJS.

In the following, we do not consider the portion of a web application that concerns the user interface, instead we focus only on the part that embodies the logic of an application.

5.1 Explicit communication and synchronization

Web applications are client-server applications that run in web browsers (client-side) and communicate with web servers (server-side). Although modern web browsers implement standards, such as WebStorage and IndexedDB, that may be used to store data on the client-side, the most prominent part of the state of an application is persisted on the server-side and transmitted to a client when it is requested.

This characteristic of web applications allows the user to access them from different locations (different web browsers on various devices) still maintaining the same state. However, that trait comes at a price for the program-

mer that has also to deal with the explicit communication between the server and the client, in addition to the main aspects of the application.

As in other client-server system, the common pattern that the programmer adopts for communication is the request-response messaging pattern: the client sends a request and the server returns a response. We are interested in how this pattern is used within a web application, after that it has been loaded by the web browser. The pattern is adopted to dynamically load resources when necessary, usually in response to user actions.

More in details, web applications, especially the single page ones, use JavaScript and specific technologies to retrieve data from the server in the background, without reload the page or transfer the control to another page. The most prominent technique currently used for these communication is Ajax. The common architecture adopts in web applications is a three-tier architecture, which is typically composed of a presentation tier, a domain logic tier and a data storage tier. For the sake of example, we consider a generic web application using Ajax and describe its flow of data.

Firstly the application makes an Ajax request to the server to obtain its persistent state, which may depend on the user too. Then, the web server receives this request and elaborates it. During this elaboration phase accesses to the underlying database can occur, in order to retrieve the information requested. Finally the server sends back the response and the client, after having received it, can load the state and shows its representation to the user. At this point, the user can interact with the application, through the UI, and changes its state. Any modification to the most prominent part of the application's state must be transmitted to the server, in order to synchronize the local version with the persistent remote one. The client sends an Ajax request with the updates, which are applied by the server, and receives the result as response.

The behaviour of a web application described until now follows the request-response pattern. However, the situation get complicated when there are more instances of the same application, maybe running on different locations, which have access to the same data. In this case, not only there is the necessity of synchronization between a client and the server, but it is also required to synchronize all the current instances of the application. Due to the characteristics of the HTTP protocol, which is stateless, and the fact that only a client can start a communication, the transmission of the updates from the server to the other instances of the application is an issue. The typical solution for this problem, when Ajax is adopted, is polling: the client periodically sends an asynchronous request to the server to check if there is any update. A more suitable solution is the already mentioned WebSocket protocol, used on top of HTTP, that provides a full-duplex communication.

The aspects of a web application that concern the communication between a client and the server and the synchronization among the different instances of the application can be made transparent to the programmer

using *Spray*.

Here and in the following sections we use a note taking application as a running example in order to demonstrate how to use SprayJS. A note is an object which has a title, a body and an optional deadline. A user can create a new note, which is added to the board containing the notes. The board is an object that contains a reference to the user and an array for his/her notes.

Listing 5.1 shows how to retrieve the board at the start of the application[1]. In this listing and in the following, `spray` is the global object that provides the interface of *Spray*. The key instruction is the `resolve` operation at line number 15, which resolve a name and return the object bound to it, i.e., the board object. After that, the `drawBoard` function implicitly loads the note objects, which are referred to the board object. Compared to what previously describe, in this example the most prominent part of the state, which is the board object, is retrieved without explicit communication.

The lines before the resolve operation retrieve the three special scopes, open them and resolve the name bound to the array containing the the users of the application. Finally, the `on` function is used to add a callback for the push event, a sub-event of change[2].

```
1  function onInit() {
2    var aScope, uScope, dScope;
3
4    aScope = spray.scope('/$APP');
5    uScope = spray.scope('/$USER');
6    dScope = spray.scope('/$DEVICE');
7    aScope.open();
8    uScope.open();
9    dScope.open();
10
11   listOfUsers = aScope.resolve('users');
12   if (listOfUsers === undefined)
13     initialiseUsers(aScope);
14
15   board = aScope.resolve('board');
16   if (board === undefined)
17     initialiseBoard(aScope, uScope);
18
19   board.notes.on('change', 'push', drawNote);
20
21   drawBoard(board);
```

[1]The variables `board` and `listOfUsers` are global.

[2]The notes field of the board object is an array containing note objects. Adding a note to the notes array is perform using the push method.

```
22  }
```

Listing 5.1 Retrieve the board.

The previous listing invokes the functions defined in Listing 5.2 and Listings 5.3 if respectively the names "users" and "board" are not bound to an object. The former listing shows how to initialise the list of user, while the latter shows how to initialise the board. The board object is created and is persisted in the application scope, whereas the notes are persisted in the user scope, because them belong to the user (any user has his/her own notes)

```
1  function initialiseUsers(appScope) {
2    listOfUsers = [];
3    spray.persist(listOfUsers, appScope);
4    aScope.bind('users', listOfUsers);
5  }
```

Listing 5.2 Initialise the list of users.

```
1  function initialiseBoard(appScope, userScope) {
2    var name, user;
3
4    name = userScope.resolve('user').name;
5    user = new User(name);
6    listOfUsers.push(user);
7    spray.deepPersist(user, appScope);
8    board = new Board(user);
9    spray.persist(board, appScope);
10   appScope.bind('board', board);
11   spray.persist(board.notes, userScope);
12   userScope.bind('notes', board.notes);
13 }
```

Listing 5.3 Initialise the board.

5.2 Sharing among users

Modern web applications allow a user to share pieces of their most prominent part of the state with other users. We can refer to a shared state, from a programmer prospective, as a complex object which is accessed by more users. On the other hand, we can adopt a conceptual point of view and call such shared state "document" or "resource". We use these terms interchangeably to indicate data with its type and structure.

In order to share a resource, the user must specify who he/she wants to share that resource with. At this point, the client sends a request to the

server that indicates which object has to be shared and the list of users. The server, after having received the request, insert the specified resource in the list of user's resources for each user in the message[3]. Then, if there are instances of the application for any of the indicated users, the server should notify them. This is another case of synchronization between the server and a client.

A calendar application and a word processor application are two types of web application that provide the possibility of sharing a resource. In the former a user can share an event with other users. In the latter a user can share a document with other users, who later edit it concurrently. With these two examples, we also want to point out that the shared document may have any size and may be arbitrary complex, as in the case of a word document.

To illustrate how to share a document among users in SprayJS we consider our note taking application. The user selects the note to share and with whom shares it. Listing 5.4 shows how to share the note with the selected users. The basic idea is to use the application scope to persist the object to share among users. In details, the note is persisted in the application scope and then it is added into the user's shared notes array and into each of the shared notes array of the specified users.

```
1  function shareNote(note, users) {
2    var aScope, i, len;
3
4    aScope = spray.scope('/$APP');
5    spray.persist(note, aScope);
6
7    board.user.shared.push(note);
8
9    len = users.length;
10   for (i = 0; i < len; i++)
11     users[i].shared.push(note);
12 }
```

Listing 5.4 Share a note.

Another interesting type of application to implement on top of SprayJS is a game. A multiplayer game is inherently shared among the players. As an example, we consider a simple game: Tic-Tac-Toe. Listing 5.5 shows the initialisation of the application[4]. The game object contains the player, the current match, if any, and the array of other pending matches. The initialisation depend on the state of the game object. If there is a current

[3]If an Access Control List (ACL) is used, it must be updated: each user of the received list is inserted into the list of permission attached to the specified object.

[4]The variables `game` and `listOfPlayers` are global.

match, it is restored and the player can continue. Otherwise the waiting room where the player can start a new game is loaded.

```
1  function onInit() {
2    var aScope, uScope, allAU, name;
3
4    aScope = spray.scope('/$APP');
5    uScope = spray.scope('/$USER');
6    aScope.open();
7    uScope.open();
8    allAU = spray.allOf(aScope, uScope);
9
10   listOfPlayers = aScope.resolve('players');
11   if (players === undefined)
12     initialisePlayers(aScope);
13
14   game = allAU.resolve('game');
15   if (game === undefined)
16     initialiseGame(aScope, uScope, allAU);
17
18   if (game.current === null) {
19     game.current.grid.on('change', undefined, drawNote);
20     drawMatch(game.current);
21   }
22   else
23     drawWaitingRoom();
24 }
```

Listing 5.5 Play Tic-Tac-Toe.

Listings 5.6 and 5.7 show respectively how to initialise the list of players and the game. In this case, we use the *allOf* scope defined by the set of the application and user scopes to persist the game object.

```
1  function initialisePlayers(appScope) {
2    listOfPlayers = [];
3    spray.persist(listOfPlayers, appScope);
4    aScope.bind('players', listOfPlayers);
5  }
```

Listing 5.6 Initialise the list of players.

```
1  function initialiseGame(appScope, userScope, allOf) {
2    var name, player;
3
4    name = userScope.resolve('user').name;
```

```
5     player = new Player(name);
6     listOfPlayers.push(player);
7     spray.deepPersist(player, appScope);
8     game = new game(player);
9     spray.persist(game, allOf);
10    allOf.bind('game', game);
11    spray.persist(game.pending, appScope);
12  }
```

Listing 5.7 Initialise the game.

A match has two players. Each of them interacts with the 3×3 grid. The grid object has been persisted in the application scope at the start of the match and it is changed at any move: a player chooses a cell of the grid and put his/her mark. The listing 5.5 shows how a move is performed[5].

```
1   function move(row, column) {
2     var grid, mark;
3
4     grid = game.current.grid;
5     mark = game.current.mark;
6
7     grid[row][column] = mark;
8     game.current.mark = (mark + 1) % 2;
9   }
```

Listing 5.8 Make a move.

The mark field of the current match is used to switch the turn to the other player, who will be notified of the change. Note that once the game object is entrusted to *Spray*, normal game logic needs not be concerned with communications to implement distribution and persistence.

5.3 Sharing among applications

Modern web app ecosystems allow an application to share pieces of their most prominent state with other applications of the ecosystems. A resource of an application may be shared with another one, which can use it later.

In these ecosystems a user has access to a set of different applications, which may manipulate the same data. Information sharing is achieved with an ad-hoc request to the server, which contains the data related to the user. In other words, a web application requests both its data and the other data needed, which logically belongs to another application.

[5]We assume that the UI does not allow the user to select a not empty cell.

As an example we can consider the Google app ecosystem, and in particular two application: Google Contacts and Gmail. The former is a contact management application, while the latter is a web mail application. A user manages its contacts with the first application: he/she can add a new contact, changes an existing one and makes groups. Then, he/she can refer to his/her list of contacts when writing a mail in Gmail. The Gmail application does not show all the contacts in the list, instead it filters the contacts with an associated email address. More in details, it shows only the name and email addresses of a contact.

In order to demonstrate how to share a resource between two applications in SprayJS, we consider the already mentioned note taking application and an application showing a set of notes sort by their deadlines. The note tacking application persists the user's notes in his/her scope, as can be observed in Listing 5.1. These notes may be retrieved later by the deadline application, as is showed in Listing 5.9[6]. This application does not show all the notes, it presents only having a deadline.

```
function onInit() {
  var aScope, uScope, allAU, notes;

  aScope = spray.scope('/$APP');
  uScope = spray.scope('/$USER');
  aScope.open();
  uScope.open();
  allAU = spray.allOf(aScope, uScope);

  board = allAU.resolve('board');
  if (board === undefined) {
    user = uScope.resolve('user')
    notes = uScope.resolve('notes');
    board = new ExpiringBoard(user, notes);
    spray.persist(board, allAU);
    allAU.bind('board', board);
  }
  if (board.notes === undefined)
    board.notes.on('change', 'push', drawExpiringNote);

  drawExpiringBoard(board);
}
```

Listing 5.9 Retrieve the user notes.

The deadline application just shows the notes. The entire application is composed of the previous listing, the function that periodically removes the

[6]The `board` variable is global.

expired notes and the function that draw the interface.

The basic idea to share an object among applications is to persist it in a scope reachable for all applications. As for the deadline example, the user scope is the obvious choice when the resource concerns the user. On the other hand, if an object is private of an application and can not be shared with other ones, it will be persisted in the application scope.

5.4 Sharing among devices

Web applications are inherently shared among more devices, due to the fact that an application can be accessed from different web browsers (running on various devices) and its state can be retrieved from there, as described in section 5.1.

In this section we do not want to demonstrate how to obtain sharing among devices in SprayJS, considering that this aspect is characteristic of web applications. Instead, our purpose is to analyse the customisation of an application based on the device and how it can be expressed in *Spray*.

The term "customisation" is used in this context not to refer to the typical modification of the graphical user interface based on the type of the device, but instead to the organisation of the information based on a priority policy that changes from a device to another one.

To the best of our knowledge, there is no web application providing customisation based on the device. However, there is a widespread application that exhibits this characteristic: Chrome, Google's web browser. A user of Chrome can save his/her bookmarks and retrieves them from both the desktop and the mobile version. From our point of view, both of the versions are installation of the same application. However, on the desktop version the desktop bookmarks are shown on top, while on the mobile version the mobile bookmarks are displayed first.

In *Spray* the customisation based on the device can be expressed. In the following, we extend our note taking application in order to have the position of a note on the board depending on the device. Thus, a note has not only a title, a body and an optional deadline, but also a position on the board. Listing 5.10 shows the function that adds a note to the board's note. The note is persisted in the user scope, while its position is persisted in the device scope.

```
function addNote(title, body, deadline) {
  var pos, note, dScope, uScope;

  pos = getNextPosition();
  dScope = spray.scope('/$DEVICE');
  spray.persist(pos, dScope);
  note = new Note(title, body, deadline, pos);
```

```
    uScope = spray.scope('/$USER');
    spray.persist(note, uScope);
    board.push(note);
  }
```

Listing 5.10 Add a note to the board.

Conclusions

"A language that doesn't affect the way you think about programming, is not worth knowing."

– Alan J.Perlis, *Epigrams on Programming*

In this thesis we have first formulated a problem concerning persistence and distribution in programming languages, explaining why we believe to be relevant. Then, we have presented the design of *Spray*, a persistence distributed heap, and presented its specification using the ASM formalism. Finally, we have sketched SprayJS, an implementation of *Spray* in JavaScript, and used it to describe some popular patterns for web applications.

Our starting point has been the lack of an abstraction which combines persistence and distribution in programming languages. Thus, we have examined how persistence and distribution have been provided in programming languages in the literature and how they are provided in the current technologies commonly used. Moreover, we have considered the CAP Theorem which concerns distributed systems and eventual consistency as an alternative consistency model.

To the best of our knowledge, there are no approaches that combine persistence and distribution at a language level and, more in particular, address these two aspects in the current scenario where the Internet is ubiquitous.

We have recognised that persistence by reachability is the only mechanism used to identify persistent objects in all the studied approaches about persistence. Furthermore, the same mechanism is adopted in the serialization techniques used in current technologies. In our work, we refer to this mechanism as "deep persistence", in contrast with the "shallow persistence" approach that we have applied.

As far as the distribution is concerned, it is obvious that if the same data is distributed and therefore replicated, the consistency among the replicas has to be guaranteed. Taken into account the CAP Theorem, we chose to prefer availability over consistency. As a result, we adopted a weak consistency model: eventual consistency. However, to remove the resolution of conflicts that happen due to concurrent updates, we have considered conflict-free replicated data types (CRDT).

In the development of this thesis, we have focused only on object-oriented

programming languages. As already mentioned, this choice is not limiting since most modern programming languages are object-oriented and provide us with the object identity. We have had to deal with the problem of How to have unique identifiers to use as identity. We chose to randomly generate such identifiers, as other real systems do.

The result of this work is the *Spray* programming, a programming paradigm for distributed application based on a persistent distributed heap. We designed how this heap is organised. The main idea in *Spray* is to arrange the persistent distributed heap in scopes. A "scope" is a special object that is bound to the underlying persistent storage and has an environment and a memory. In other words, a scope binds names to identifiers and contains the state of objects.

The shallow persistence approach is a characteristic of *Spray* that we believe is important to remark. The mechanism used to identify persistent objects contributes to the novelty of our work. Shallow persistence lets to split an object on multiple scopes. Furthermore, an object can be persisted in more than one scope. Combining these two features we can have different states of an object that may differ only in the value of a field, depending on which scopes are open.

Using the ASM formalism, we presented the specification of *Spray*. It consist of the specification of the operations on a scope, the extension for dealing with generic objects that have been persisted, the operations on a storage and the communication among storages.

We believe that the *Spray* programming paradigm can simplify the development of applications concerning persistence and distribution. To this purpose, we developed a proof-of-concept implementation of the specification for web applications in JavaScript. That implementation is called SprayJS. As already mentioned, JavaScript has been chosen because it is the lingua franca of the Web, it is flexible enough to let to extend the language at runtime and has the advantage that an object contains both fields and methods.

Finally, we have validated the *Spray* programming paradigm through examples. We examined some popular patterns for web applications and described their implementation in SprayJS. In these examples we have shown how to use the different characteristics of our work.

Further development

The purpose of this thesis is the design and specification of the *Spray* programming and its persistent distributed heap, our abstraction that combines persistence and distribution. However, there are other aspects of our idea that have not been considered in this work and may be taken into account in a future work. Furthermore, there are some ideas that may be developed starting from this thesis. Specifically:

- It would be possible to develop a fully functional implementation of SprayJS that manages properly all the built-in objects of JavaScript. Moreover, it would be possible to implement *Spray* in other languages in order to develop also desktop and mobile application using such paradigm.

- This work does not concern aspects related to security and access control. We would study how to transmit operations over a network ensuring protection and how to provide access control for scope objects.

- We implemented *Spray* in a prototype-based language, however most of the object-oriented programming languages are class-based. In these languages the definition of the methods is in the class and it is separated from the fields, which belong to the object. It would be possible to implement *Spray* in a class-based language, dealing with the issues that would arise. Moreover, there is the problem of the schema evolution to consider.

- Our approach has the implicit purpose of hiding to the programmer files, file systems, databases and explicit communication. Databases are traditionally used to manage and querying huge volumes of data. Relational databases are a particular class of databases that are quite general as far as their use is concerned. To increase their performance, queries are generally optimised before being executed and specific indices can be created. In *Spray* instead, how objects are retrieved from the persistent storage depend on their structures. In other words, there is a shift from a generic approach that can optimise to a ad-hoc approach where the programmer has to design an object structure based on his/her needs. We think that may be interesting comparing the performance of these two different approaches.

- Finally, it would be possible to design a type system ensuring that any type is a conflict-free replicated data type. Although CRDTs are not a general approach, because not all types can be described as a CRDT, a restricted type system supporting only CRDT may be useful to improve and integrating the implementation of *Spray*.

Bibliography

[Abelson et al., 1996] Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, second edition.

[Arnold and Gosling, 1996] Arnold, K. and Gosling, J. (1996). *The Java Programming Language*. Addison-Wesley.

[Atkinson et al., 1983] Atkinson, M. P., Bailey, P. J., Chisholm, K., Cockshott, W. P., and Morrison, R. (1983). An Approach to Persistent Programming. *Comput. J.*, 26(4):360–365.

[Atkinson et al., 1989] Atkinson, M. P., Bancilhon, F., DeWitt, D. J., Dittrich, K. R., Maier, D., and Zdonik, S. B. (1989). The Object-Oriented Database System Manifesto. In *DOOD*, pages 223–240.

[Atkinson et al., 1982] Atkinson, M. P., Chisholm, K., and Cockshott, P. (1982). PS-algol: An Algol with a Persistent Heap. *SIGPLAN Not.*, 17(7):24–31.

[Atkinson et al., 1996a] Atkinson, M. P., Daynès, L., Jordan, M. J., Printezis, T., and Spence, S. (1996a). An Orthogonally Persistent Java. *SIGMOD Rec.*, 25(4):68–75.

[Atkinson and Jordan, 1999] Atkinson, M. P. and Jordan, M. J. (1999). Issues Raised by Three Years of Developing Pjama: An Orthogonally Persistent Platform for Java. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings.*, pages 1–30.

[Atkinson and Jordan, 2000] Atkinson, M. P. and Jordan, M. J. (2000). *A Review of the Rationale and Architectures of PJama - a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform*, volume 2000-90 of *SMLI TR*. Sun Microsystems Laboratories.

[Atkinson et al., 1996b] Atkinson, M. P., Jordan, M. J., Daynès, L., and Spence, S. (1996b). Design Issues for Persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *POS*, pages 33–47.

[Bal et al., 1990] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. (1990). Experience with Distributed Programming in ORCA. In *1990 Internation Conference on Computer Languages, March 12-15 1990, New Orleans, Louisiana, USA*, pages 79–89.

[Bal et al., 1992] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. (1992). Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Trans. Software Eng.*, 18(3):190–205.

[Black et al., 1986] Black, A. P., Hutchinson, N. C., Jul, E., and Levy, H. M. (1986). Object Structure in the Emerald System. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings.*, pages 78–86.

[Black et al., 1987] Black, A. P., Hutchinson, N. C., Jul, E., Levy, H. M., and Carter, L. (1987). Distribution and Abstract Types in Emerald. *IEEE Trans. Software Eng.*, 13(1):65–76.

[Bläser, 2006] Bläser, L. (2006). A Programming Language with Natural Persistence. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 637–638, New York, NY, USA. ACM.

[Bläser, 2007] Bläser, L. (2007). Persistent Oberon: A Programming Language with Integrated Persistence. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, pages 71–85.

[Boehm, 2005] Boehm, H. (2005). Threads cannot be implemented as a library. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 261–268.

[Börger and Stärk, 2003] Börger, E. and Stärk, R. F. (2003). *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer.

[Bray, 2014] Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, Internet Engineering Task Force.

[Bray et al., 2008] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report, World Wide Web Consortium.

[Brewer, 2000] Brewer, E. A. (2000). Towards Robust Distributed Systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA. ACM.

[Brewer, 2012] Brewer, E. A. (2012). Pushing the CAP: strategies for consistency and availability. *IEEE Computer*, 45(2):23–29.

[Cardelli, 1994] Cardelli, L. (1994). Obliq: A language with Distributed Scope. Technical report, Digital Equipment Corporation, Systems Research Center.

[Cardelli, 1995] Cardelli, L. (1995). A Language with Distributed Scope. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 286–297.

[Cattell, 2010] Cattell, R. (2010). Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27.

[Chang et al., 2006] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. (2006). Bigtable: A Distributed Storage System for Structured Data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218.

[Cisternino et al., 2005] Cisternino, A., Cazzola, W., and Colombo, D. (2005). Metadata-Driven Library Design. In *Proceedings of Library-Centric Software Design Workshop*, LCSD'05.

[Codd, 1970] Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387.

[Cook and Rosenberger, 2005] Cook, W. R. and Rosenberger, C. (2005). Native Queries for Persistent Objects, A Design White Paper.

[Coulouris et al., 2011] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition.

[Dahl et al., 1968] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1968). Some Features of the SIMULA 67 Language. In *Proceedings of the Second Conference on Applications of Simulations*, pages 29–31. Winter Simulation Conference.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150.

[Dearle et al., 1989] Dearle, A., Connor, R. C. H., Brown, F., and Morrison, R. (1989). Napier88 - A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages, 4-8 June, 1989, Salishan Lodge, Gleneden Beach, Oregon*, pages 179–195.

[Dearle et al., 1996] Dearle, A., Hulse, D., and Farkas, A. (1996). Operating System support for Java. In *Proceedings of the First International Workshop on Persistence and Java*.

[Dearle et al., 2009] Dearle, A., Kirby, G. N. C., and Morrison, R. (2009). Orthogonal Persistence Revisited. In *Object Databases, Second International Conference, ICOODB 2009, Zurich, Switzerland, July 1-3, 2009. Revised Papers*, pages 1–22.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220.

[Derrett et al., 1985] Derrett, N., Kent, W., and Lyngbæk, P. (1985). Some Aspects of Operations in an Object-Oriented Database. *IEEE Database Eng. Bull.*, 8(4):66–74.

[Dittrich, 1986] Dittrich, K. R. (1986). Object-oriented Database Systems (Extended Abstract): The Notions and the Issues. In *Proceedings on the 1986 International Workshop on Object-oriented Database Systems*, OODS '86, pages 2–4, Los Alamitos, CA, USA. IEEE Computer Society Press.

[ECMA, 2011] ECMA (2011). *ECMA-262: ECMAScript Language Specification*. Ecma International.

[ECMA, 2013] ECMA (2013). *ECMA-404: The JSON Data Interchange Format*. Ecma International.

[Fette and Melnikov, 2011] Fette, I. and Melnikov, A. (2011). The WebSocket Protocol. RFC 6455, Internet Engineering Task Force.

[Gabbrielli and Martini, 2010] Gabbrielli, M. and Martini, S. (2010). *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer.

[Gilbert and Lynch, 2002] Gilbert, S. and Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59.

[Gosling et al., 1996] Gosling, J., Joy, W. N., and Jr., G. L. S. (1996). *The Java Language Specification*. Addison-Wesley.

[Gray, 1981] Gray, J. (1981). The Transaction Concept: Virtues and Limitations (Invited Paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154.

[Härder and Reuter, 1983] Härder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317.

[Haridi et al., 1997] Haridi, S., Roy, P. V., and Smolka, G. (1997). An overview of the design of Distributed Oz. In *Proceedings of the 2nd International Workshop on Parallel Symbolic Computation, PASCO 1997, July 20-22, 1997, Kihei, Hawaii, USA*, pages 176–187.

[Hericko et al., 2003] Hericko, M., Juric, M. B., Rozman, I., Beloglavec, S., and Zivkovic, A. (2003). Object serialization analysis and comparison in Java and .NET. *SIGPLAN Notices*, 38(8):44–54.

[Huang and Luo, 2013] Huang, Y. and Luo, T. (2013). NoSQL Database: A Scalable, Availability, High Performance Storage for Big Data. In *Pervasive Computing and the Networked World - Joint International Conference, ICPCA/SWS 2013, Vina del Mar, Chile, December 5-7, 2013. Revised Selected Papers*, pages 172–183.

[International Telecommunication Union, 2012] International Telecommunication Union (2012). Information technology - Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers. X. 667, Telecommunication Standardization Sector of ITU.

[Ireland et al., 2009] Ireland, C., Bowers, D., Newton, M., and Waugh, K. (2009). A Classification of Object-Relational Impedance Mismatch. In *The First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDS 2009, Gosier, Guadeloupe, France, 1-6 March 2009*, pages 36–43.

[Jordan, 1996] Jordan, M. (1996). Early Experiences with Persistent Java.

[Jordan and Atkinson, 1998] Jordan, M. and Atkinson, M. (1998). Orthogonal Persistence for Java - A Mid-term Report.

[Jordan, 2004] Jordan, M. J. (2004). A Comparative Study of Persistence Mechanisms for the Java Platform. Technical report, Mountain View, CA, USA.

[Jul et al., 1988] Jul, E., Levy, H. M., Hutchinson, N. C., and Black, A. P. (1988). Fine-Grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.*, 6(1):109–133.

[Khoshafian and Copeland, 1986] Khoshafian, S. N. and Copeland, G. P. (1986). Object Identity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 406–416, New York, NY, USA. ACM.

[Kulkarni et al., 2007] Kulkarni, D., Bolognese, L., Warren, M., Hejlsberg, A., and George, K. (2007). LINQ to SQL: .NET Language-Integrated Query for Relational Data. `https://msdn.microsoft.com/en-in/library/bb425822.aspx`.

[Leach et al., 2005] Leach, P., Mealling, M., and Salz, R. (2005). A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, Internet Engineering Task Force.

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

[Li, 1986] Li, K. (1986). *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, New Haven, CT, USA. AAI8728365.

[Li and Hudak, 1989] Li, K. and Hudak, P. (1989). Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4):321–359.

[Maier, 1989] Maier, D. (1989). Why Isn't There an Object-Oriented Data Model? In *IFIP Congress*, pages 793–798.

[Maier et al., 1985] Maier, D., Otis, A., and Purdy, A. (1985). Object-Oriented Database Development at Servio Logic. *IEEE Database Eng. Bull.*, 8(4):58–65.

[Marquez et al., 2000] Marquez, A., Blackburn, S., Mercer, G., and Zigman, J. N. (2000). Implementing Orthogonally Persistent Java. In *Persistent Object Systems, 9th International Workshop, POS-9, Lillehammer, Norway, September 6-8, 2000, Revised Papers*, pages 247–261.

[Microsoft Corporation, 2013] Microsoft Corporation (2013). LINQ (Language-Integrated Query). `https://msdn.microsoft.com/en-us/library/bb397926.aspx`. Accessed 15 May 2015.

[Morrison et al., 2000] Morrison, R., Connor, R. C., Cutts, Q. I., Kirby, G. N., Munro, D. S., and Atkinson, M. P. (2000). The Napier88 Persistent Programming Language and Environment.

[Moss and Hosking, 1996] Moss, J. E. B. and Hosking, A. L. (1996). Approaches to Adding Persistence to Java.

[Node.js Foundation, 2015] Node.js Foundation (2015). Node.js. `https://nodejs.org`. Accessed 1 Jul 2015.

[Oracle Corporation, 2005] Oracle Corporation (2005). Java Object Serialization Specification (version 6.0). `http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`. Accessed 11 May 2015.

[Oracle Corporation, 2013] Oracle Corporation (2013). JSR-338: Java Persistence 2.1. `https://jcp.org/en/jsr/detail?id=338`.

[Pokorný, 2011] Pokorný, J. (2011). NoSQL Databases: a step to database scalability in Web environment. In *iiWAS'2011 - The 13th International Conference on Information Integration and Web-based Applications and Services, 5-7 December 2011, Ho Chi Minh City, Vietnam*, pages 278–283.

[Pritchett, 2008] Pritchett, D. (2008). BASE: an acid alternative. *ACM Queue*, 6(3):48–55.

[Roy et al., 1997] Roy, P. V., Haridi, S., Brand, P., Smolka, G., Mehl, M., and Scheidhauer, R. (1997). Mobile Objects in Distributed Oz. *ACM Trans. Program. Lang. Syst.*, 19(5):804–851.

[Sadalage and Fowler, 2012] Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition.

[Schwarzkopf et al., 2013] Schwarzkopf, M., Grosvenor, M. P., and Hand, S. (2013). New wine in old skins: the case for distributed operating systems in the data center. In *Asia-Pacific Workshop on Systems, APSys '13, Singapore, Singapore, July 29-30, 2013*, pages 9:1–9:7.

[Scott, 2009] Scott, M. L. (2009). *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition.

[Shapiro et al., 2011a] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011a). A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, INRIA.

[Shapiro et al., 2011b] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011b). Conflict-free replicated data types. [Research Report] RR-7687, INRIA.

[Shapiro et al., 2011c] Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. (2011c). Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 386–400.

[Stonebraker, 1986] Stonebraker, M. (1986). The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9.

[Stonebraker, 2010] Stonebraker, M. (2010). SQL databases v. NoSQL databases. *Commun. ACM*, 53(4):10–11.

[Strauch, 2012] Strauch, C. (2012). NoSQL Databases. Lecture of Selected Topics on Software-Technology Ultra-Large Scale Sites. Technical report, Stuttgart Media University.

[Sun Microsystems, Inc., 2000] Sun Microsystems, Inc. (2000). The Forest Project. `http://web.archive.org/web/20041013072348/http://www.sunlabs.com/research/forest/`.

[Takasaka, 2005] Takasaka, S. (2005). Survey of Persistence Approaches. Master's thesis, Royal Institute of Technology/Stockholm University in collaboration with Swiss Federal Institute of Technology Zurich - ETH.

[Tanenbaum, 1993] Tanenbaum, A. S. (1993). Distributed operating systems anno 1992. What have we learned so far? *Distributed Systems Engineering*, 1(1):3–10.

[Tanenbaum, 1995] Tanenbaum, A. S. (1995). *Distributed Operating Systems.* Prentice Hall.

[Tanenbaum, 2008] Tanenbaum, A. S. (2008). *Modern operating systems (3. ed.).* Pearson Education.

[Tansey and Eli, 2008] Tansey, W. and Eli, T. (2008). Efficient automated marshaling of C++ data structures for MPI applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12.

[Torgersen, 2006] Torgersen, M. (2006). Language integrated query: unified querying across data sources and programming languages. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 736–737.

[Torgersen, 2007] Torgersen, M. (2007). Querying in c#: how language integrated query (LINQ) works. In *Companion to the 22nd Annual ACM*

SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, pages 852–853.

[Vogels, 2008] Vogels, W. (2008). Eventually Consistent. *ACM Queue*, 6(6):14–19.

[Zope Foundation, 2015] Zope Foundation (2015). ZODB - a native object database for Python. `http://www.zodb.org`. Accessed 22 May 2015.