ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea (Magistrale) in Ingegneria e Scienze Informatiche

Bridging Logic Programming with platform-independent distributed services

Tesi di laurea in
Sistemi Distribuiti

Relatore
Prof. Giovanni Ciatto

Correlatore
Prof. Andrea Omicini

Candidato
Lorenzo Osimani

Seconda Sessione di Laurea
Anno Accademico 2022-2023

Abstract

Researchers have always been interested in combining the logic paradigm with other technologies to create new hybrid approaches. Until now, however, it was difficult to make languages like Prolog interact with external entities. In this thesis, we propose a solution that enables a logical solver to delegate the execution of requests to remote service, possibly implemented in languages different from that of the solver. Accordingly, we provide a prototype that leverages the concepts of logic primitives and the As-A-Service model and we implement a concrete example that merges symbolic AI with sub-symbolic AI.

Ai miei nonni, che continuano a guidarmi nella mia vita

Acknowledgements

Ringrazio innanzitutto i professori del corso di studio di Ingegneria e Scienze Informatiche, che mi hanno seguito in questi intensi anni di formazione e hanno contribuito alla crescita della mia figura professionale. In particolare ringrazio il professor Giovanni Ciatto, per avermi accompagnato in questi mesi di lavoro di tesi con serietà, pazienza e l'entusiasmo che ha saputo trasmettermi.

Ringrazio poi tutti i miei amici, sia quelli le cui strade si sono allontate dalla mia che quelli che camminano ancora oggi al mio fianco, perchè ognuno di loro è stato un pezzo fondamentale del percorso della mia maturità umana.

Infine ringrazio la mia famiglia, soprattutto mio padre Maurizio, mia madre Elisabetta e mia sorella Maria Chiara per essere stati dei pilastri che mi hanno sostenuto sia nei momenti di gioia che in quelli di difficoltà.

Contents

List of Figures

Listings

Chapter 1

Introduction

Nowadays many topics of research are being investigated in the computer science field. Innovations that combine different technologies are continuously developed, bringing new potential features into the real world. Some of these areas of study, however, are unable to interact easily with the rest and are often the cause of *technological silos*, isolated points where some data is segregated and difficult to exchange with the overall system.

Symbolic AI [11] is one of these areas. It refers to all methods in artificial intelligence that are based on high-level symbolic and human-readable representations of problems, logic, and search. These methodologies are, for example, exploited in Prolog, a logic programming language that defines logical solvers and queries them with the resolution of predicates.

Many research groups have investigated possible hybrid frameworks that integrate symbolic AI with other technologies [12, 3, 13]. Yet, due to symbolic AI characteristics and isolated nature, they often have defined theories of potential developments but they lack a concrete *how* that enables the interaction.

An interesting topic that researchers have tried to incorporate with symbolic AI is sub-symbolic AI [14]. It concerns all AI methodologies where information is generated from relations of high complexity. These relations are often formalized with functions that map the input to the output data or the target variables. An example is Artificial Neural Networks (ANN), which are widely used in Machine Learning. By uniting the two branches of AI, the resulting technology would inherit the advantages of both techniques.

This thesis proposes a concrete approach to bridge symbolic AI, in particular the Prolog environment, with any other technology, including sub-symbolic AI. The idea is to apply the As-A-Service model [9] and shift the resolution of predicates into a distributed view. During the execution, the logic engine will interact with remote services and delegate part of the computation externally. These services, furthermore, can be defined with Interface Description Languages, such as gRPC [24] and proto buffers [7], which are language-independent. Thus, a solver can interact with a service to exploit mechanisms and languages previously unavailable for the resolution of queries, without leaving the logic realm.

More technically, in this thesis we define the interface of a generic service and the communication protocol with a logical solver using IDL. We then prototype the system by extending the 2p-kt [4] framework and test its characteristics. Finally, to demonstrate the potentialities of this approach, we provide an example of integration between symbolic and sub-symbolic AI by implementing the machine learning API described in [6] for Prolog.

Thesis Structure. Accordingly, the remainder of this thesis is structured as follows. Chapter 2 delves into the primary concepts and technologies that make up the core elements of the project and that will be used throughout the entire paper. Chapter 3 presents the design process that defined the system structure and behavior. Chapter 4 showcases the actual implementation of the system, and contains examples of code snippets of the prototype. Chapter 5 discusses how the prototype was validated and provides an example of how it can be applied in a real context. Finally, Chapter 6 concludes this thesis by summarising its main contribution.

Chapter 2

Background

This chapter will list the ideas, mechanisms, and technologies that will be recurring in the rest of the paper and that have been used during the project development.

2.1 Logic Resolution And Logic Programming

When tackling a problem in the symbolic AI field, such as the demonstration of a theorem or the analysis of some data with Artificial Intelligence, there are various approaches available. One of these methods is *logic resolution* [21]. Logic resolution is an inference rule for automated reasoning in propositional logic, and it is based on the *principle of refutation*, where a statement is proven by demonstrating that its negation leads to a contradiction.

In propositional logic, knowledge can be expressed using Horn clauses [23], which are logic formulae in the form of disjunctions of predicates. An example of a clause is

```
H :- B1, ..., Bn.
```

which indicates that the head `H` is true if the body composed of the literals `B1, ..., Bn` is also true. Clauses can also represent a number of predicates to be proven, in which case are known as *goals*. By applying the resolution rule to them we can determine their satisfiability in a particular context.

The resolution process is composed of three main steps:

Unification: It consists of finding a substitution, called *most general unifier*, that can make two literals identical [18]. For instance, to match the literals `p(X)` and `p(a)`, a valid substitution would be `X = a`.

Resolution: It involves combining two clauses by selecting two complementary literals, one from each clause, and unifying them. The resolved literals are later removed from both the original clauses, which are then merged to create a resolvent that serves as an intermediate step.

Simplification: In this step redundant literals or tautologies are eliminated from the obtained clause.

If the final outcome of the resolution process is an empty clause, it indicates that the initial goal presents a contradiction and is therefore not true.

Logic resolution is commonly used in automated theorem proving and logical reasoning, by recursively applying a specific resolution rule and producing a new clause that follows logically from the initial one. This method enables the derivation of new logical conclusions or the verification of satisfiability or unsatisfiability of logical formulas.

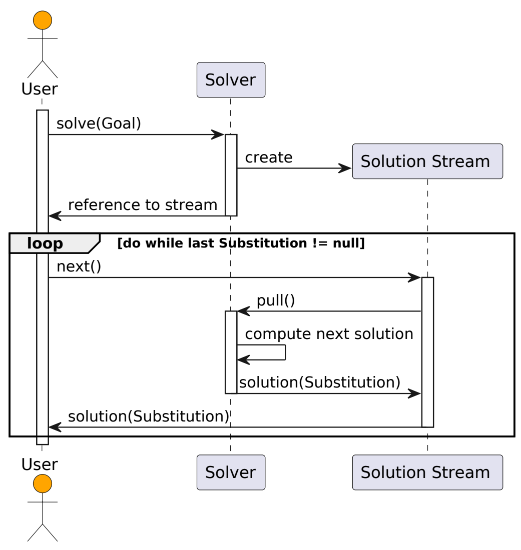
From the concepts of logic resolution, a new programming approach has been developed with time, called *Logic Programming* [1]. Logic Programming is a declarative programming paradigm used mainly in the field of AI. Instead of a sequence of commands and instructions, a program is composed of sentences and rules in logical form. A problem is described as a set of clauses named *knowledge base* (KB) that represents the various entities of its domain, their relations, and their constraints. Goals can then be passed to the program in the form of queries, and the system will attempt to apply resolution and inference on the provided facts and rules to generate logical deductions or solutions.

A key feature of Logic Programming is its support for *non-determinism* and *backtracking*. During the resolution of a query, in fact, the program can explore alternative paths and find multiple solutions by systematically exploring the solution space. If a chosen course leads to a contradiction or a dead end, the system can backtrack and explore the remaining available branches until all possibilities are analyzed.

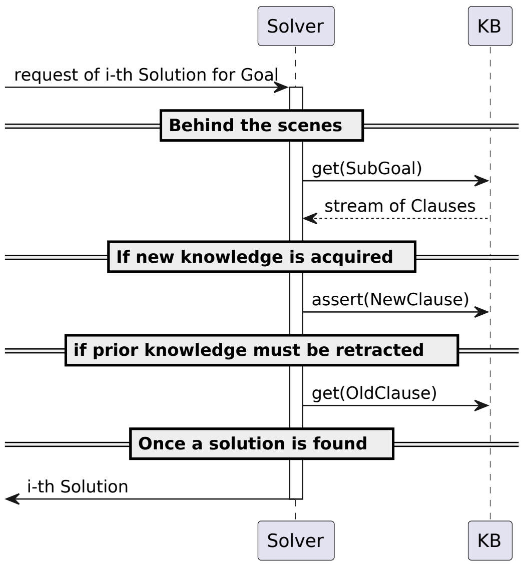
2.2 Prolog

One of the most well-known declarative programming languages is Prolog[16], which stands for PROgramming in LOGic. Prolog programs consist of Horn clauses that can be categorized as either *facts* or *rules*. Facts are fixed entities in the problem's knowledge base, while rules define the logical relationships between those entities. In the resolution process, this data is used to reach new logical conclusions.

Prolog is commonly used in data-driven AI due to its capacity to investigate vast amounts of information [2]. In particular, the clauses in a knowledge base can be general and may not always be accurate, providing a level of uncertainty that is useful in this field of study. To compute solutions of a query, the Prolog



(a) Stream-oriented interaction mode



(b) Interaction among a logic solver and its KB

Figure 2.1: Figures of interaction modes between logic solvers and users or KB from [5]

system employs a form of depth-first search called *backward chaining*, constructing a tree where each node represents one of the logical resolution steps. A solution can either succeed, fail, or encounter an error that stops its execution. Queries can also contain variables, which are represented by literals beginning with a capitalized letter. Any successful returned solution will be a possible substitution of variables that makes the query true within that particular program's knowledge base.

A reactive logic entity that interacts with a knowledge base and can resolve queries requested by a user is called solver *à la Prolog*. Its knowledge base can be either static, dynamic, or a combination of both, depending on how it can be modified during the execution of goals. As detailed in [5], backtracking enables a solver to return solutions of a query on a lazy stream. The stream generates a new value every time the user requests a new solution until no more are available. Thus, a Prolog solver acts as both a producer and consumer of streams of solutions. Figure 2.1 provides an example of the interaction between users, solvers, and knowledge bases.

In order to find new solutions, a solver can consume streams of data produced either by its knowledge base or by an external source. This external source can take the form of a *primitive*, which is a logical predicate that enables data manipulation and interaction with the Prolog environment. Usually a primitive is implemented

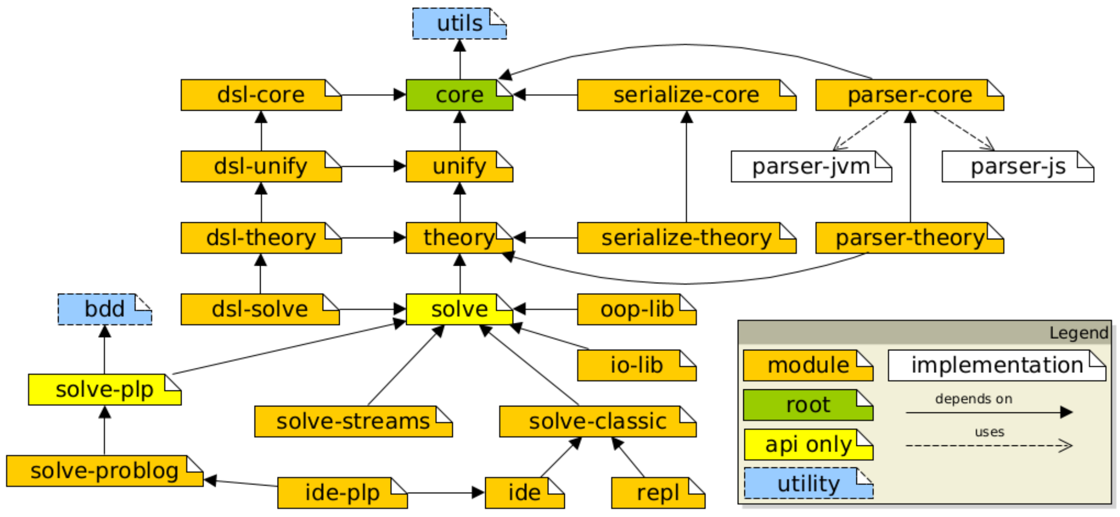


Figure 2.2: 2P-Kt modules structure

in a low-level programming language. Examples of built-in primitives are `is/2`, which is used for arithmetic calculations, and `write/1`, which writes on the output buffer. When a solver queries a primitive, it sends a request with any necessary arguments and variables. The primitive then generates one or more responses and possibly side effects that will be applied to the solver environment.

2.2.1 2P-Kt

Learning Prolog itself is relatively easy, and it is rather simple to understand. However, due to the syntax required when defining a program, it may not be user-friendly in particular contexts. To address this issue, a research team has developed *2p-kt* [4], a framework written in Kotlin that wraps the core mechanisms of Prolog in a multi-paradigm environment while also expanding its functionalities. The framework includes various modules that are incrementally interdependent, each introducing new features for symbolic manipulation and reasoning. Some of the primary modules are:

`:core` → contains the logic terms defined in Prolog rewritten in Kotlin. The most generic type of logical entity is `Term`.

`:solve` → a generic API to instantiate and use solver entities for the resolution of logic queries, extended by some specific implementations in other modules like `:solve-classic`.

`:parser-core` and `:parser-theory` → used respectively to parse `Terms` and `Theories` into serialized items.

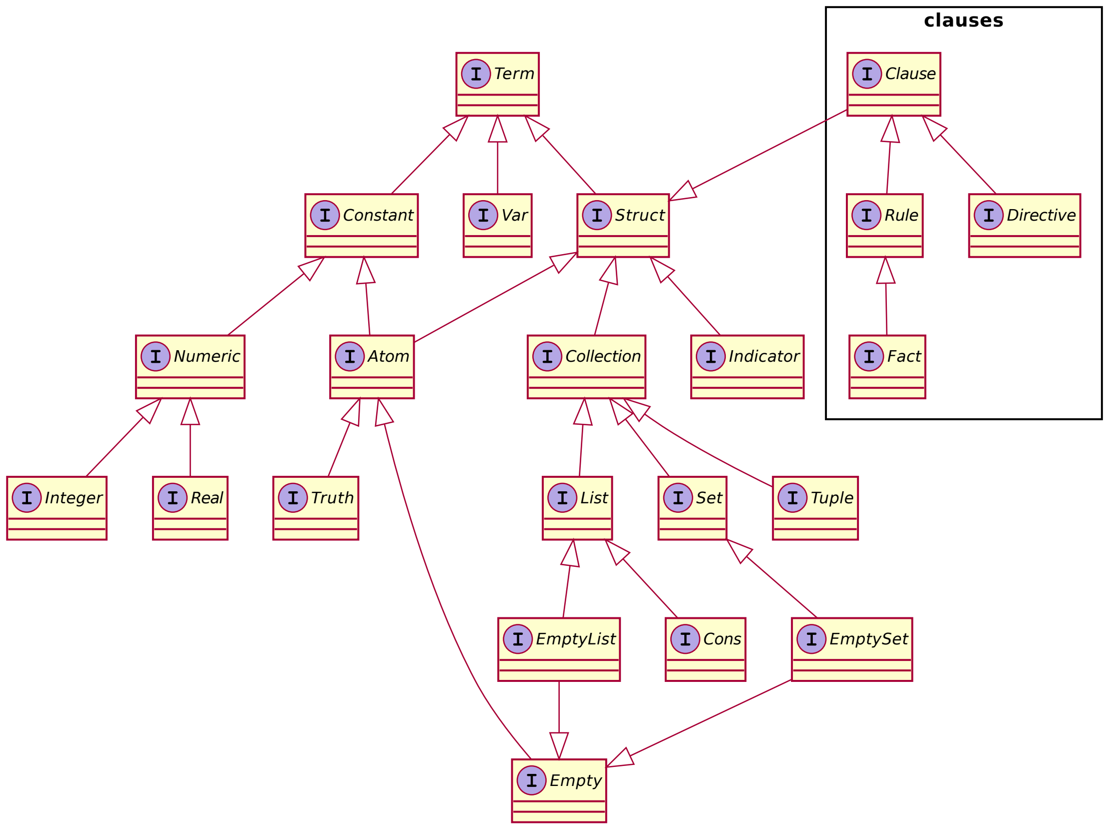


Figure 2.3: Hierarchy of Prolog elements in the `:core` module

`:io-lib` → a module that contains an API of operations usable in a Prolog program to interact with input and output channels.

The framework is a *general-purpose open ecosystem*. It virtually supports several platforms, like JVM, JS, and Android, due to its multi-platform nature. Moreover, it is very lightweight and minimal since it only leverages the Kotlin standard library. Using the module `:solve`, a user can create a solver entity with custom initialization parameters, like the knowledge bases, I/O channels, operators, and libraries. These libraries are a collection of functions, operators, and clauses that can extend the syntax and functionalities supported in a solver.

Thanks to the structure of 2p-kt it is possible to easily implement custom primitives written in Kotlin. Primitives can, in fact, be considered as I/O operations that delegate the computation to some external entity and that return *lazily* the stream of calculated values. By assigning them to a *signature*, composed of the function name and its arity, they can be inserted in a library and later imported into a solver. This way the possible procedures are no longer limited by the structures of Prolog and they can be defined externally. At the moment, however, the only supported language is Kotlin, the same one used to implement the 2p-kt framework.

2.3 Prolog Applications In The Real World

In the modern world, Prolog is used in various contexts. Some examples of applications are the following:

Automated Reasoning: [17] Prolog's primary usage is to automatically derive logical conclusions from a given set of facts and rules. This is a major functionality applied in the fields of AI and theorem proving.

Database Querying: [15] Because of its pattern-matching capabilities, Prolog allows developers to define logical queries and retrieve data from databases, including complex relationships and retrieval criteria.

Constraint Satisfaction Problems (CSP): [19] Prolog has built-in constraint logic programming features, making it suited to approach constraint satisfaction problems like resource allocation or scheduling.

Natural Language Processing (NLP): [10] NLP problems encompass applications such as comprehending and parsing natural language sentences. By defining grammar, syntactic and semantic rules, Prolog can be used for tasks like information retrieval and question answering from texts written in natural languages.

Prolog performances and applications are, however, limited to the logic paradigm. Researchers have theorized the possible results of combining symbolic AI with other fields of study [12, 3, 13], yet we still lack an actual bridge that connects Prolog with the functionalities offered by those technologies.

A research topic that would offer interesting advantages if merged with Prolog is *sub-symbolic AI* [14]. Unlike symbolic AI methods, sub-symbolic AI attempts to replicate the intricate network of neurons in the human brain and learns by defining complex functions that map input data to target variables. Sub-symbolic AI includes various statistical learning methods, such as Bayesian learning, deep learning, backpropagation, and genetic algorithms. The symbolic approach is best suited for small, precise data and often requires human intervention during the learning process. In contrast, the sub-symbolic techniques can handle large and noisy datasets effectively and adapt autonomously. A hybrid method would thus be able to both learn from the environment and reason the results.

2.3.1 Python and Prolog

Naturally, Python[8] comes to mind when talking about AI, as it is widely used in the field of machine learning. Python is a high-level, general-purpose programming language and it supports multiple programming paradigms, including structured, object-oriented, and functional programming. Python was also designed to be highly extensible via modules and thus it provides an *extensive ecosystem* of libraries.

NumPy, pandas, scikit-learn, and TensorFlow are only some instances of the frameworks available. By exploiting them, Python can run complex operations of data manipulation and define machine learning models with few and simple lines of code. Python's flexibility and integration capabilities allow users to seamlessly integrate AI algorithms with components of data processing pipelines, making it exceptionally well-suited for real-world applications.

Using Python functionalities in a Prolog-like environment could allow users to combine automated reasoning and sub-symbolic AI approaches. For example, the authors of [6] present the structure of an API in Prolog, where the methods enable users to load datasets, pre-process data, select and define predictors' models (in particular neural networks), train these predictors, use them for inference of data and validate the results without leaving the logic realm. However, it is only a theoretical prototype and lacks an actual implementation.

2.4 The As-A-Service Model

Evolving monolithic structures into the distributed paradigm has become a trend to improve the performance and quality of new technologies. One of the leading models for defining complex distributed systems is the As-A-Service model [9], which consists of taking a certain resource, from a simple physical infrastructure to software, and making it available remotely: users can request the service's capabilities by communicating on the network. This model introduces many advantages like scalability, fault tolerance, improved performance, and relieving the client of the physical allocation of the necessary resources. However, it also complicates the development process by introducing critical issues like security concerns and network dependence.

Services can be defined through Interface Descriptor Languages (IDL) [22]. An IDL is a language used to describe structured data types and interfaces in a *language-independent* way, thus allowing a program or object written in one language to communicate with another program written in an unknown language. Two examples of IDL are Protocol Buffers and gRPC.

2.4.1 Protocol Buffers

Protocol Buffers [7], also referred to as *protobuf*, is a language-neutral, platform-neutral extensible mechanism for serializing structured data. It was developed by Google and it's used for data serialization and designing communication protocols between systems, for example, in distributed environments.

A `.proto` file is defined as a set of messages written in a specific syntax that can contain scalar types, strings, nested and repeated messages as fields. By using this structured model, data is serialized in a compact binary format, smaller in size compared to formats like XML or JSON. This makes Protocol Buffers more efficient for data transmission over the network due to fewer storage requirements.

The basic syntax of a `.proto` file is very simple, but it can be extended with custom options and features, for instance, services that define Remote-Procedure-Calls (RPC) interfaces. These files can later be used to automatically generate the relative classes in many programming languages, including C++, Java, Python, Go, Ruby, C#, JavaScript, and more. Moreover, protobuf supports the versioning of message definitions with mechanisms of backward and forward compatibility.

2.4.2 gRPC

gRPC [24], on the other hand, is a modern open-source high-performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking, and authentication. gRPC is based on the *remote procedure call paradigm*, where a client can invoke methods on the server application as if they were local function calls. The complexities of network communication are hidden from the user, keeping its usage transparent.

The system supports various types of calls, from *unary calls*, where a single request from the client corresponds to a single response from the server, to *streaming calls*, where both clients and servers send independently continuous streams of messages on the communication channel.

The services interfaces are defined with Protocol Buffers, allowing gRPC to support multiple languages and enabling interoperability across different language implementations; moreover, due to protobuf, the message structure is strongly typed. The default transport protocol used is HTTP/2, which allows multiplexing, flow control, and header compression, while also supporting many security mechanisms like Transport Layer Security (TLS) and the possibility to add interceptors and middleware.

Thus, by utilizing protobuf and gRPC we can define effective service interfaces and message protocols without having to concern ourselves with which languages are used to implement each distributed entity.

Chapter 3

Design

3.1 Objective and Requirements

We previously stated that Prolog is considered a technological silo due to its inability to interact and communicate easily with entities and functionalities from paradigms that transcend the logic realm. This thesis proposes to bridge that gap, by integrating the concept of Prolog primitives with the As-A-Service model.

As described in Chapter 2, a primitive is a logical predicate defined by a functor, an arity representing the number of arguments it accepts, and an elaboration process that lazily returns the values in a sequence. Figure 3.1 illustrates the usage of a primitive with a client-server metaphor. When the user queries a goal to the solver involving a primitive, it sends a request. The request may include the arguments provided for the computation and a reference to the *execution context*, which correspond to the state of the solver and can be used by the primitive to query the resolution of additional sub-goals.

The primitive returns a lazy sequence of responses, each containing a generated value, a list of side effects that modify the solver's state, or an error encountered during the resolution. The responses are later elaborated to generate the solutions for the user.

Figure 3.1 highlights the compatibility of the primitive mechanism with the As-A-Service model. A primitive can be seen as an external service that provides a method to *lazily* return solutions. The interface of the service and the message protocol can be defined with an IDL, thus allowing the service to be implemented in any compatible language.

The idea of this thesis is to develop a distributed system that enables a solver to exploit various language-specific functionalities while remaining in a logic environment. In other words, we want to design a platform-independent primitive service interface and its message protocol used to communicate with a logic solver

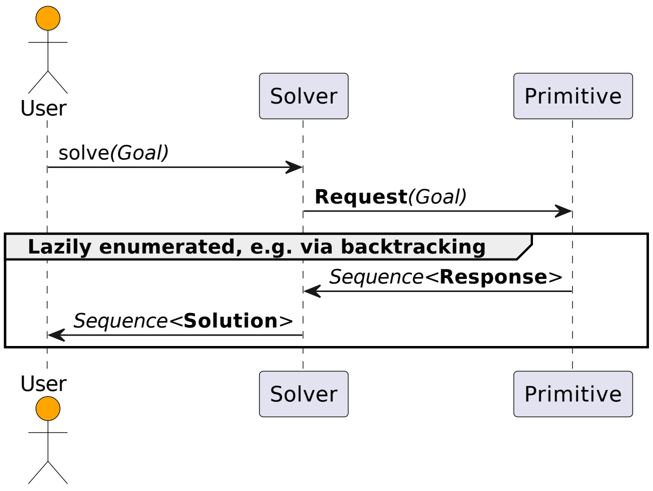


Figure 3.1: Primitive representation as a client-server metaphor taken from [4]

à la Prolog.

Functional Requirements

- A solver must be able to remotely call a primitive service during the resolution process, optionally providing arguments, and receive a lazy sequence of solutions.
- A primitive service should be able to query additional information from a solver when computing a request, such as the current state of its execution context or the logic resolution of sub-goals.

Non Functional Requirements

- The server interface must be generic enough to allow users to implement any kind of primitives, that may vary in name, computation process, number and types of arguments.
- The server interface must be platform-independent.
- The introduction of distributed entities should not significantly worsen the performances in comparison to a Prolog execution of primitives.
- A primitive service should be able to handle multiple requests concurrently.

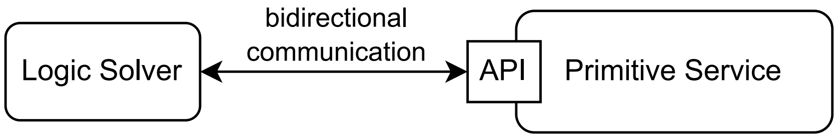


Figure 3.2: Diagram of the system structure

Technological Requirements

- The service interface and message protocol must be implemented using an IDL, possibly gRPC and protobuf.
- The distributed system should be compatible with the methods and the structures present in the framework 2p-kt.

3.2 General System Structure

The generic architecture of the system is composed of the two entities illustrated in figure 3.2, which are a solver acting as the client and a primitive service. When a solver connects with the service and requests a new computation process, a stream of messages is exchanged between the two parts until no more solutions can be generated or the client interrupts the connection. We will from now on refer to the stream of messages and the data relative to a single invocation of the primitive with the term *session*.

3.2.1 Server Interface

While connected, the client may use the service API to invoke the following operations:

- Request the initialization of a new computation process, eventually by providing arguments.
- Request to compute, if available, a new value of a specific computation process.
- Send the service the result of a sub-task, such as part of the execution context state or the resolution of a sub-goal.

On the other hand, the service can perform these operations:

- Generate and send the client a new value relative to a specific computation process.

- Query the client with a sub-task and await the response.
- Send the solver a side effect to apply to its execution context.

It's worth noticing that it is impossible to define a priori a linear execution order of these operations since it depends on the actual computation process of the remote primitive. For example, the number of sub-tasks requested before the service replies with a generated value is unknown. To maintain the service's implementation flexible, the interface is defined with these characteristics:

- an attribute that represents the remote primitive functor and arity, called `signature`.
- an invocation that creates a bidirectional channel where all the messages are sent, named `callPrimitive`.

The invocation method `callPrimitive` is the core of the interface, as it encompasses all possible interactions during a session. Its communication channel is composed of two streams of messages, one for each sender.

3.2.2 Message Protocol

The message protocol is designed as a hierarchy of structured entities divided by sender, following the characteristics of the service interface.

Common Messages

`StructMsg:` It represents an element of the Prolog syntax, such as a variable, a function, a constant, or a fact.

`ArgumentMsg:` It is used to wrap `Struct` elements in arguments of functions or substitutions included in a solution.

`ResponseMsg:` It contains one solution and the side effects generated either by the service or by the client during a sub-goal execution.

`SolutionMsg:` It holds data about a solution, like if it succeeded and with which values, failed, or encountered an error.

`ErrorMsg:` It represents an error encountered during the computation of a solution. A hierarchy of more specific error messages that extends this structure is available.

`SideEffectMsg:` It represents a side effect that must be applied to the execution context. As for the errors, the different types of side effects are organized in a hierarchy based on this structure.

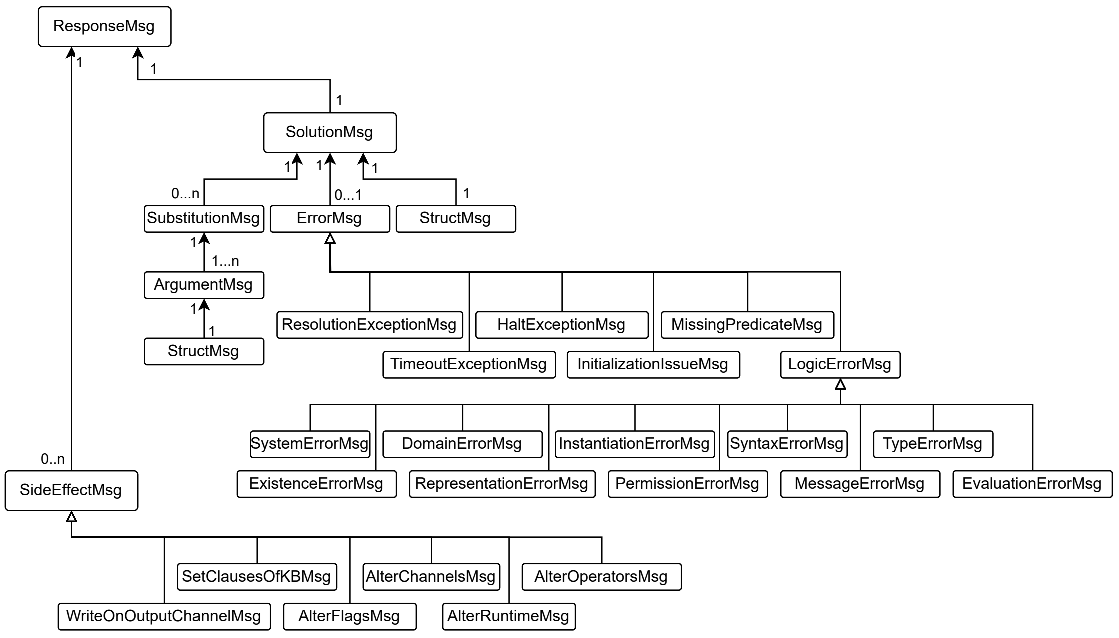


Figure 3.3: General hierarchy of common messages

Service-Specific Messages

`NextSolution`: An empty message sent to request the generation of a new solution.

`SubRequestMsg`: This message is sent by the service to query a sub-task to the client. The more specific messages associated with the available operations are the following:

- `SubSolveRequest`: It requests the resolution of a sub-goal to the solver-client.
- `ReadLineMsg`: It queries the reading of a line from an input channel of the solver.
- `InspectKbMsg`: It is sent to inspect the knowledge bases of the solver, possibly by providing filters for the clauses.
- `GenericGetMsg`: It requests one of the elements of the current execution context.

Client-specific Messages

`RequestMsg` : This message is sent by the client to start a session with the service; it contains the data about the primitive called, the arguments provided, and

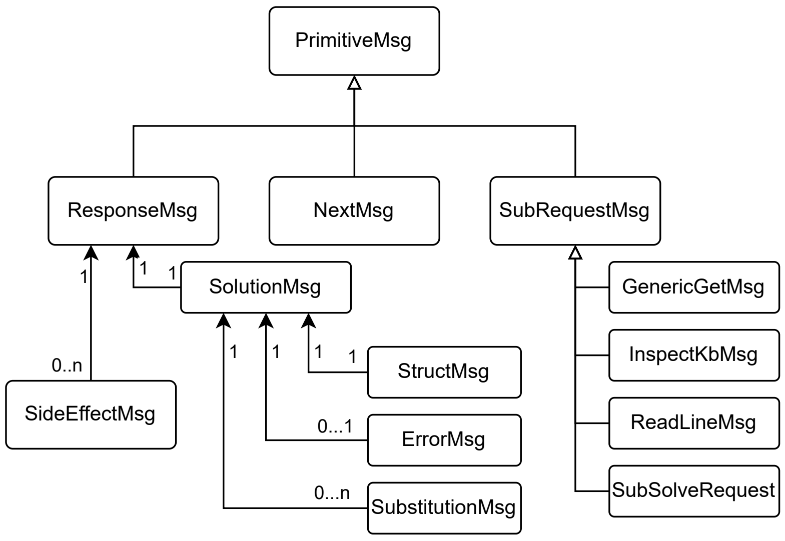


Figure 3.4: General hierarchy of service-specific messages

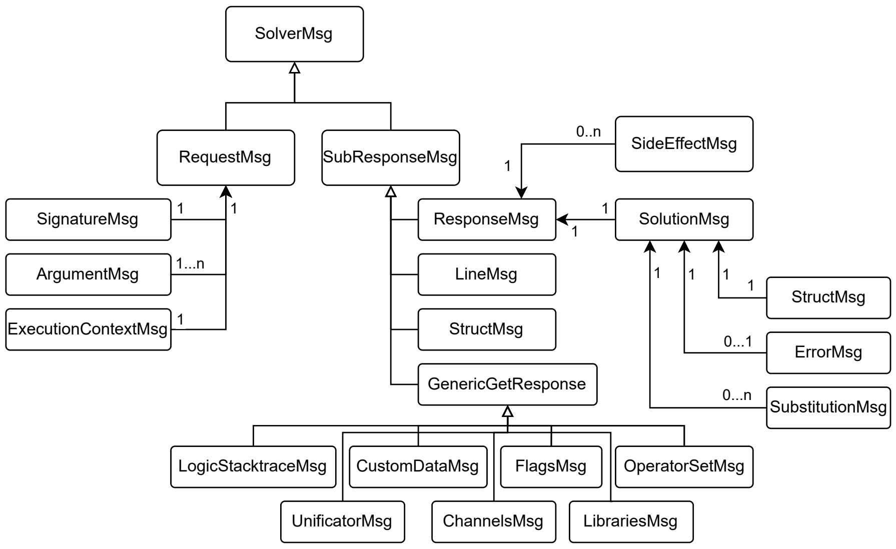


Figure 3.5: General hierarchy of client-specific messages

the basic information about the current execution context of the solver.

`SignatureMsg` : It contains the functor and arity of the primitive called.

`ExecutionContextMsg` : It includes the basic elements of the execution context, such as the current procedure evaluated and the substitutions of variables already known.

`SubResponseMsg` : It is sent as the response to a sub-task requested by the primitive service during its execution. It may contain:

- `LineMsg:` It includes a string read from an input channel of the solver.
- `StructMsg:` It represents one of the clauses of the knowledge base, and it is returned when the service queries the inspection of the current state of the static or dynamic KB.
- `GenericGetResponse:` It contains one of the following elements of the execution context:
  - `LogicStacktraceMsg:` The current logic stack trace of execution.
  - `CustomDataMsg:` A key-value store that may keep any sort of data and that can be used during computations.
  - `UnificatorMsg:` A set of variables and the values to which they can be unified.
  - `LibrariesMsg:` Libraries cannot be serialized and sent on the network since they are composed of executables. Therefore only the names and the methods' signatures of the libraries imported by the solver are in this structured message.
  - `FlagsMsg:` A set of flags and their values used in the solver's configuration.
  - `OperatorSetMsg:` The set of operators usable to parse queries of the solver.
  - `ChannelsMsg:` The names of the I/O channels.

To avoid the high cost of sending the entire execution context in one message and the additional exchanges needed to keep it up-to-date, we chose to provide the server with various sub-tasks that allow the remote primitive to retrieve each element individually, one at a time. This partitioning approach is also applied when handling other large groups of data, such as theories and knowledge bases, which are sent through multiple messages as lazy streams of clauses.

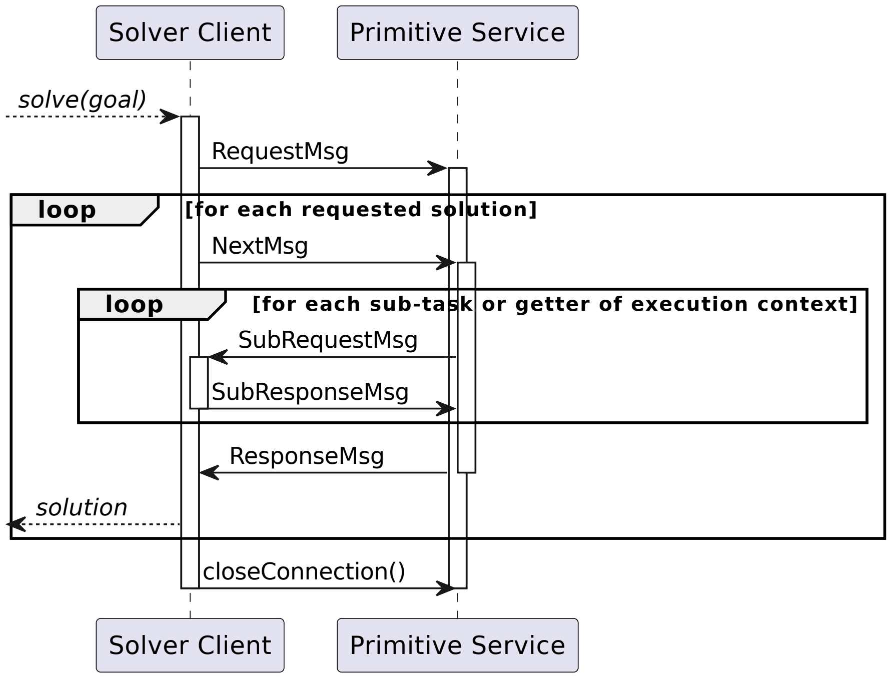


Figure 3.6: Diagram of a generic messages exchange

3.3 Interaction

As previously stated, during the invocation of a remote primitive all the communication happens in a bidirectional stream. Figure 3.6 displays the general structure of message exchange between the two distributed entities, enhancing the schema presented in 3.1. Since the actual resolution procedure of a primitive depends on its implementation, the order of messages may vary accordingly: for instance, multiple sub-tasks may occur simultaneously, or be resolved across different solutions' dispatch. Nonetheless, a `ResponseMsg` is sent on the stream for each `NextMsg`, and for each `SubRequestMsg` there should be a `SubResponseMsg`, in whichever order they may be exchanged.

3.4 Distributed Entities Structure

Once the general behavior of the system is defined, we delve in more detail into the components of each distributed entity of the system.

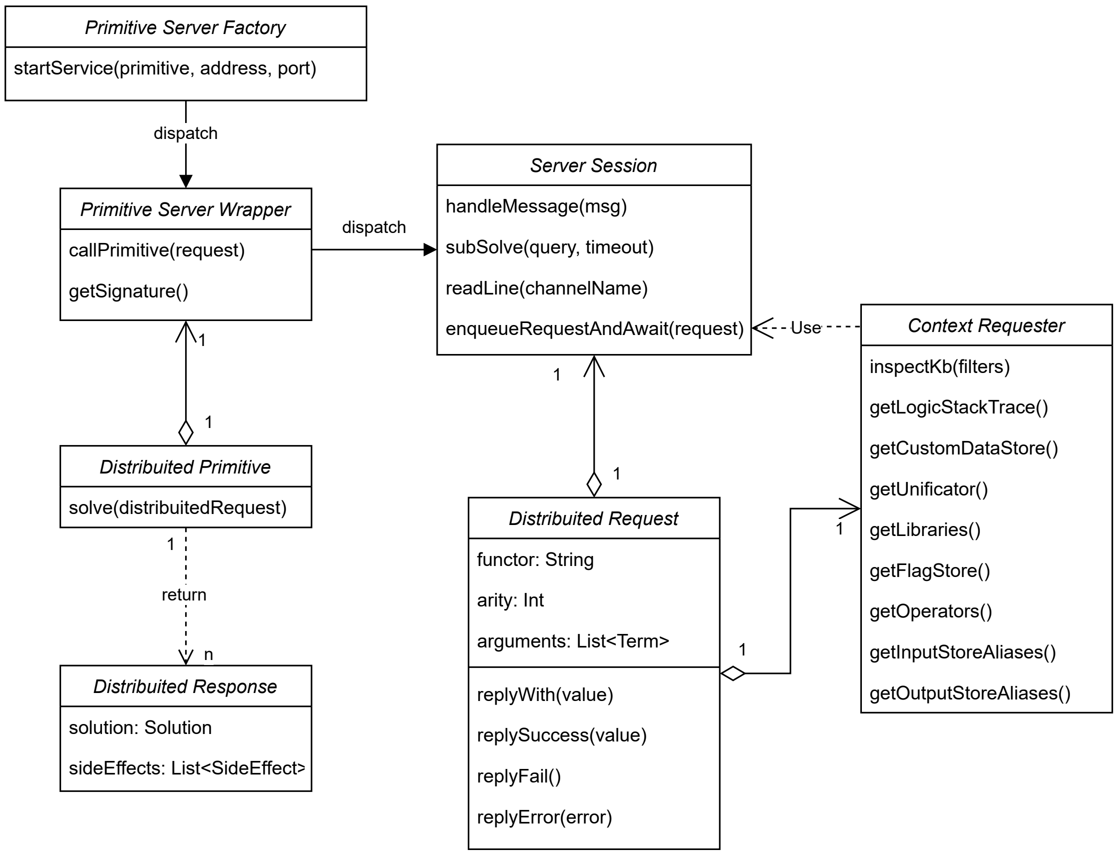


Figure 3.7: Server's generic structure

3.4.1 Primitive Service

When designing the structure of a service, we must consider that depending on which language is actually used, the language paradigm may vary and different syntax mechanisms may be available. However, it is still possible to define a general layout, shown in 3.7, that every implementation should follow to its best abilities. Its components are the following:

Distributed Request It represents a request received from the client: it contains the arguments provided by the client solver to be used during the computation. It also exposes the methods to issue a sub-task or to retrieve any element of the execution context.

Distributed Response Its instances are what compose the sequence of values generated by the primitive. It can contain a substitution of the variables in the query or a possible error encountered during the resolution. Moreover, it includes the eventual side effects that will be applied to the solver.

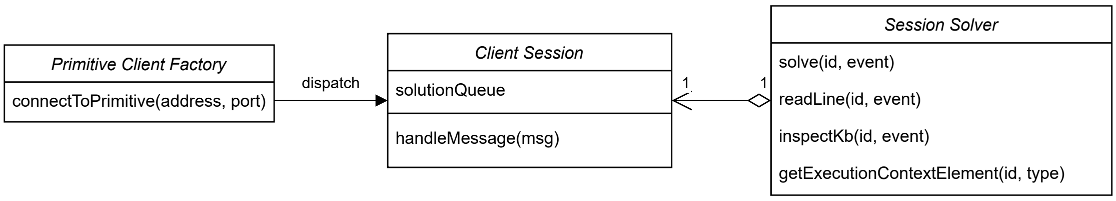


Figure 3.8: Client's generic structure

Distributed Primitive It holds the resolution process of any primitive-as-a-service. It presents a higher-order function that takes a Distributed Request and returns a sequence of Distributed Responses. During resolution, additional data can be retrieved from the client through the methods exposed by the Distributed Request object. The function must employ a mechanism to lazily return the values generated by the primitive. For example, in the basic implementation of primitives in 2p-kt, written in Kotlin, the keyword `yield` should be used, which computes the first value and then suspends the execution until the next one is requested.

Server Session It is the core of the server and it contains all the logic that keeps track of the session's data and handles the messages received from the client. It also exposes the methods to build all the types of replies and it sends them on the stream of communication.

Context Requester It acts as a remote execution context, where all its fields are mapped to dispatch the corresponding `GenericGetMsg` to the client.

Primitive Server Wrapper It implements the service API previously defined. Thus it is the active component that listens for the messages arriving, redirects them to the right component, initializes sessions, and sends all the responses to the clients.

Primitive Server Factory Given the implementation of the distributed primitive and which port it will listen onto, its method produces the instances of servers and starts them.

3.4.2 Client

The client extends a standard solver, introducing structures that enable the interaction with the service API.

Client Session It contains the client that actually connects with the service. When prompted, it requests new solutions to the remote primitive and handles the stream of messages received, executing the sub-tasks when necessary.

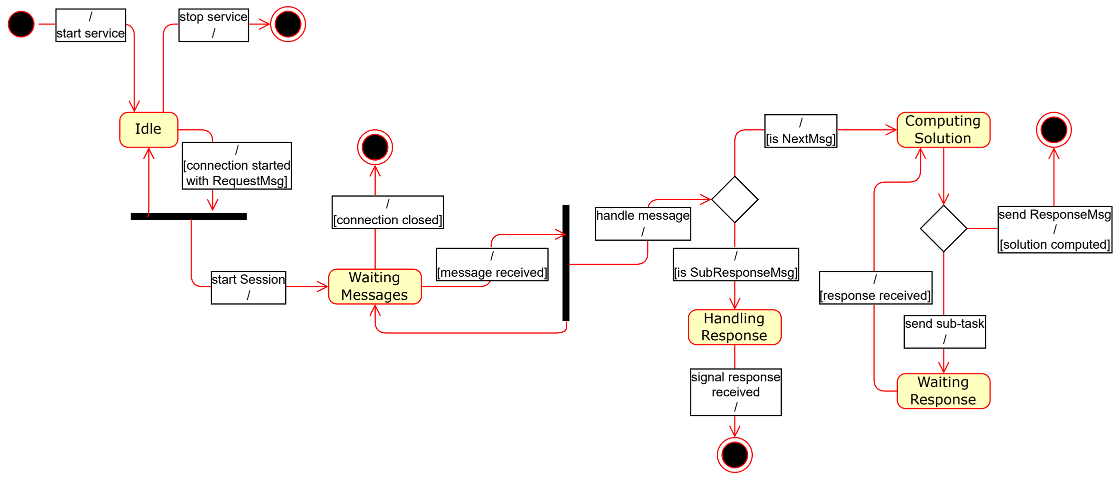


Figure 3.9: Control flow diagram of a service

Session Solver It keeps track of the information about the current session with the primitive. It has data on any ongoing sub-task and contains a copy of the solver's execution context: this copy is used to reply to the sub-requests of the service.

Primitive Client Factory It presents the method that, given which address to connect to and which port, starts the connection with the remote service and returns the relative Client Session instance.

3.5 Distributed Entities Behavior

Both service and client are structured around the idea of managing multiple concurrent tasks. On both sides, the main task listens for new messages, and a new task is generated for each message received. The components can therefore handle multiple messages simultaneously, allowing single tasks to await additional data without blocking the whole component.

3.5.1 Primitive Service

A primitive service instance initially listens for clients' connections and `RequestsMsg`. Once a connection is established, a new task initializes the session and waits for messages in an async manner on the opened channel stream. When a message is received, another task is created to handle it. If additional data is required from the client, the corresponding message is sent on the communication stream and the current task awaits the response. Otherwise, if the message received contains the response of an ongoing sub-task, it unlocks the latter by signaling the received value. When a solution is finally computed, a `ResponseMsg` is sent on the stream.

3.5.2 Client Proxy

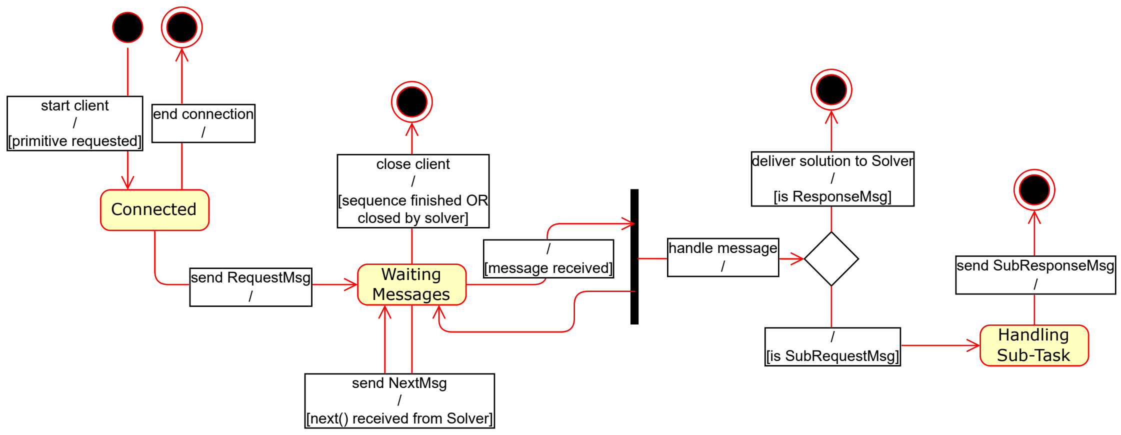


Figure 3.10: Control flow diagram of a client

The actual client is instantiated when a solver queries the primitive. After establishing the connection with the service, it waits on the channel stream for messages. If a `ResponseMsg` is received, it is made available for consumption by the solver. If any other message is received instead, it is handled by dispatching the requested data on the stream. The Session Solver also maintains data for any ongoing sub-task during the execution, like the values already generated for a sub-solve and the sequence to retrieve new ones if available. Once all the values of the primitive have been received or the solver terminates the sequence, the client closes the connection.

Chapter 4

Implementation

In this chapter, we will describe the prototype's implementation, which extends 2p-kt with a new module called `:primitives-aas`. During the development, we employed gRPC, protobuf, and the other modules already present in the Prolog framework. While the client is written in Kotlin, the same language used for 2p-kt, the service structure can be implemented in many different languages. For the sake of demonstrating the service's platform independence, we developed its prototype both in Kotlin and Python.

4.1 The Service Interface

As explained previously, gRPC allows the definition of different types of remote procedure calls in `.proto` files. The simpler one of these is the unary RPC, where a single message from the client corresponds to a single reply from the service. Another type available is the bidirectional streaming RPC, on which the messages are sent on two separate streams, one for each side: any call that is initiated by the client creates a different channel that is managed concurrently wrt the others.

The service interface is defined in protobuf, as shown in Listing 4.1, using the proto3 syntax. It is composed of two methods, a bidirectional streaming RPC named `callPrimitive` which handles the messages exchange of a resolution's session, and a unary RPC called `getSignature` that returns the primitive's signature.

Listing 4.1: Service's interface implementation in proto3

```
service GenericPrimitiveService {
  rpc callPrimitive(stream SolverMsg) returns (stream PrimitiveMsg) {}
  rpc getSignature(EmptyMsg) returns (SignatureMsg) {}
}
```

Messages are also defined in `.proto` files as small logical records of information containing a series of name-value pairs called fields. These fields can be primitive types such as int, enum, and bool, more complex structures like lists and maps, or user-defined types. An example of a message is shown in Listing 4.2, which contains the definition of `SolutionMsg`.

Listing 4.2: Example of message implementation in proto3

```
message SolutionMsg {
  SolutionType type = 1;
  StructMsg query = 2;
  map<string, ArgumentMsg> substitutions = 3;
  optional ErrorMsg error = 4;
  bool hasNext = 5;
  enum SolutionType {
    SUCCESS = 0;
    FAIL = 1;
    HALT = 2;
  }
}
```

Thanks to a special gRPC plugin, from the `.proto` files it is possible to generate automatically code that includes the gRPC client and server classes, as well as the methods for populating, serializing, and retrieving the message types. Thus, it is feasible to extend these files and implement in detail our system's functionalities and behavior without worrying about the low-level coding of establishing and managing the connection.

In order to easily transform the 2p-kt entities in messages and vice versa, we also defined a collection of serializers. The serialization methods are implemented as extensions of the original classes, such as in Listing 4.3.

Listing 4.3: Serialization method of a Prolog Term

```
fun Term.serialize(): ArgumentMsg {
    val builder = ArgumentMsg.newBuilder()
    when (this) {
        is Var -> builder.setVar(this.name)
        is Truth -> builder.setFlag(this.isTrue)
        is Numeric -> builder.setNumeric(this.decimalValue.toDouble())
        is Atom -> builder.setAtom(this.value)
        is Struct -> builder.setStruct(this.serialize())
    }
    return builder.build()
}
```

Listing 4.4: Kotlin implementation of the `callPrimitive()` method

```kotlin
class PrimitiveServerWrapper private constructor( ... ):
    GenericPrimitiveServiceGrpc.GenericPrimitiveServiceImplBase() {

  override fun callPrimitive(responseObserver: StreamObserver<
      PrimitiveMsg>): StreamObserver<SolverMsg> {
    return object : StreamObserver<SolverMsg> {

      private var session: ServerSession? = null

      override fun onNext(value: SolverMsg) {
        when (session) {
          null ->
            if (value.hasRequest()) {
              session = ServerSession.of(primitive, value.request,
                  responseObserver)
            } else { ... }
          else -> {
            executor.execute {
              session!!.handleMessage(value)
            }
          }
        }
      }
  ...
```

4.2 Implementation of a Primitive-As-A-Service

4.2.1 In Kotlin

Following the structure described in 3.4.1, we implemented in Kotlin the service architecture inside the `:primitives-aas` module.

The `PrimitiveServerWrapper` class is implemented by extending the server code generated by the gRPC plugin and therefore it presents the methods defined in the service interface, as shown in Listing 4.4. In particular, the bidirectional streaming RPC `callPrimitive(...)` requires as argument the `StreamObserver` object used by the server to dispatch replies with the `onNext()` method, and returns the `StreamObserver` object that intercepts the client messages. If a `RequestMsg` is received on the stream, a session is initialized. Otherwise, a new task is spawned to handle the message.

The handling of messages is delegated to the `ServerSession` class in Listing 4.5. Depending on the content received, the task elaborates it accordingly:

- if a `NextMsg` has been received, it iterates over the primitive's sequence of solutions replying with a new value. If the sequence has already been termi-

Listing 4.5: Kotlin implementation of message handling in `ServerSession`

```kotlin
override fun handleMessage(msg: SolverMsg) {
    /** Handling Next Request */
    if (msg.hasNext()) {
        val response = try {
            stream.next().serialize(stream.hasNext())
        } catch (_: NoSuchElementException) {
            request.replyFail().serialize(false)
        }
        responseObserver.onNext(
            PrimitiveMsg.newBuilder().setResponse(response).build()
        )
        if (!response.solution.hasNext) responseObserver.onCompleted()
    }
    /** Handling SubRequest Event */
    else if (msg.hasResponse()) {
        ongoingSubRequests.find { it.id == msg.response.id }.let {
            it?.signalResponse(msg.response)
        }
    }
    /** Throws error if it tries to initialize again */
    else if (msg.hasRequest()) { ... }
}
```

nated, the service replies with a failure. Additionally, if no more solutions are available to be calculated, the connection is closed with the stream's `onCompleted()` method.

- if the message is a `SubResponseMsg`, the task signals the relative pending sub-task with the received value.

The basic elements of the primitive used in the service, such as the request and response object, recall the implementations already present in the 2p-kt framework. However, they have been revised in a new version to comply with the message protocol's structure and the distributed nature of the system.

For instance, the solutions' sequence is generated by the `DistributedPrimitive` entity, a higher-order function that provides the actual implementation of the resolution process and is defined when initializing the service instance. The interface to be extended is Listing 4.6. Its `solve()` method accepts a `DistributedRequest` object and returns a sequence of `DistributedResponse` that can be iterated lazily.

An example of `DistributedPrimitive` implementation is Listing 4.7, a rewriting of the primitive `natural/1` which either asserts if the argument provided is a natural number or, in case of a variable argument, returns a sequence of incremental numbers.

Listing 4.6: Interface of a `DistributedPrimitive` in Kotlin

```kotlin
fun interface DistributedPrimitive {

    fun solve(request: DistributedRequest): Sequence<DistributedResponse>

    companion object { ... }
}
```

Listing 4.7: Implementation of the `nt/1` distributed primitive in Kotlin

```kotlin
DistributedPrimitiveWrapper("nt", 1) { request ->

    when (val arg1: Term = request.arguments[0]) {
        is Var ->
            generateSequenceOfNaturals().map { request.replySuccess(
                Substitution.of(arg1, it)) }
        is Integer ->
            sequence{
               yield(request.replyWith(arg1 >= 0)
            }
        else ->
            sequenceOf(request.replyFail())
    }
}
```

The keyword `yield` and the `Sequence` type are used to obtain the lazy nature of a primitive so that the computation process blocks after each generated value until the next one is requested.

In order to implement the request for a sub-task, we defined the `SubRequestEvent` interface, shown in Listing 4.8. A `SubRequestEvent` is composed of the message sent to the client to query the task, a randomly generated id, and two methods, `awaitResult()` and `signalResponse()`, used respectively to await the value requested and to signal the response's reception by manipulating an internal `Deferred` object.

An example of a sub-task type is the `SubSolve` task, which queries the client solver a Prolog sub-goal and returns the sequence of computed solutions. Listing 4.9 and Listing 4.10 present the implementation of the two methods used to handle a single response of a sub-solve request.

When a sub-task is requested, the function `enqueueRequestAndAwait()`, illustrated in Listing 4.11, is called, in which these steps will be taken in order:

- It adds the `SubRequestEvent` object to the set of ongoing sub-tasks.
- It sends the corresponding `SubRequestMsg` to the client.

Listing 4.8: `SubRequestEvent`'s implementation in Kotlin

```kotlin
interface SubRequestEvent {

    val message: PrimitiveMsg

    val id: String

    fun signalResponse(msg: SubResponseMsg)

    fun awaitResult(): Any?
}
```

Listing 4.9: Kotlin implementation of the `awaitResult` method in a `SubSolve` request

```kotlin
override fun awaitResult(): DistributedResponse {
    val response = runBlocking {
        result.await()
    }
    hasNext = response.solution.hasNext
    return response.deserializeAsDistributed()
}
```

Listing 4.10: Kotlin implementation of the `signalResponse` method in a `SubSolve` request

```kotlin
override fun signalResponse(msg: SubResponseMsg) {
    if (msg.hasSolution()) {
        this.result.complete(msg.solution)
    } else {
        throw IllegalArgumentException("The message received is not of a
            SubSolve")
    }
}
```

Listing 4.11: Implementation of `enqueueRequestAndAwait()` method in Kotlin

```kotlin
override fun enqueueRequestAndAwait(
        request: SubRequestEvent
    ): Any? {
        ongoingSubRequests.add(request)
        responseObserver.onNext(request.message)
        return request.awaitResult().also {
            ongoingSubRequests.remove(request)
        }
    }
```

Listing 4.12: Implementation of `solve/2` remote primitive in Kotlin

```kotlin
1  DistributedPrimitiveWrapper("solve", 1) { request ->
2      request.subSolve(request.arguments[0].castToStruct()).map {
3          if (it.solution.isYes) {
4              request.replySuccess(it.solution.substitution.castToUnifier()
                   )
5          } else if (it.solution.isNo) {
6              request.replyFail()
7          } else {
8              request.replyError(it.solution.exception!!)
9          }
10     }
11 }
```

- It awaits the relative `SubResponseMsg`. As the answer with the correct ID is received on a separate task, the method is awakened and the value obtained is returned.

- The `SubRequestEvent` is then removed from the set of active sub-tasks.

To request a sub-task, the corresponding method must be called in the primitive's resolution process on the `DistributedRequest` object. An example is shown in line 2 of Listing 4.12 with the method `subsolve()`. The same listing also illustrates an implementation of the `solve/1` primitive which queries the client solver with the sub-goal provided as the function's argument.

Finally, to run a service we call the method present in the `PrimitiveServerFactory` object and illustrated in Listing 4.13. It takes as arguments the primitive's implementation coupled with its signature and the port where the service will be deployed. The body of the function then runs these instructions:

- It creates an `Executor` that will manage the threads used by the service to handle the received messages and to complete the various tasks of the computation processes.

Listing 4.13: Implementation of `startService()` method in Kotlin

```kotlin
fun startService(
    primitive: DistributedPrimitiveWrapper,
    port: Int = 8080
): Server {
    val executor = Executors.newCachedThreadPool()
    val service = PrimitiveServerWrapper.of(primitive, executor)
    val genericPrimitive = ServerBuilder.forPort(port)
        .addService(service)
        .executor(executor)
        .build()
    genericPrimitive!!.start()
    Runtime.getRuntime().addShutdownHook(
        Thread {
            genericPrimitive.shutdownNow()
        }
    )
    return genericPrimitive
}
```

- It instantiates the `PrimitiveServerWrapper`, and uses it to launch the actual server.
- It adds a shutdown hook to close the server in case of interruptions.
- It returns the server's object.

4.2.2 In Python

A primitive service can be implemented in other languages, such as Python. Apart from some minor differences due to the different syntax, the structure and implementation are similar to the ones in Kotlin.

The first main variation is in the `callPrimitive()` function, as shown in Listing 4.14. The server classes generated by gRPC, in fact, use the `Generator` type to manage the streams of messages. Thus we utilize the keyword `yield` not only to generate the primitive values but also to send replies to the client.

The method `messageHandling()` is defined within the body of `callPrimitive()`. It processes the received messages and adds the resulting replies to the `queue` object in a manner similar to the Kotlin implementation. This process is run concurrently by the service Executor, while the main control flow waits for items to be added to the `queue` and sends them through the stream using `yield`.

The same structure of distributed core elements is maintained, for example by developing the `DistributedPrimitive` class as shown in Listing 4.15. The

Listing 4.14: Python implementation of the `callPrimitive` RPC

```
def callPrimitive(self, request_iterator, context: Context):
  queue = Queue[primitivesMsg.GeneratorMsg]()

  def messageHandling(): [...]

  self.executor.submit(messageHandling)

  context.add_callback(lambda: queue.put(None))

  while context.is_active():
    msg: primitivesMsg.GeneratorMsg = queue.get()
    if(msg != None):
      yield msg
    else:
      context.cancel()
```

Listing 4.15: Abstract class of `DistributedPrimitive` in Python

```
class DistributedPrimitive(ABC):
  @abstractmethod
  def solve(self, request: DistributedRequest) -> Generator[
      DistributedResponse, None, None]:
    pass
```

`solve()` function is defined with the `@abstractmethod` tag, which will be overridden by the concrete implementations of the primitives.

To provide an example, we chose to implement the primitive `nt/1` in Python too. The code is illustrated in Listing 4.16, and as it can be observed there aren't major differences apart from the usage of generators instead of the `Sequence` type.

4.3 The Client's Implementation

In a local environment, a primitive is made available by adding it to a library and then by importing the latter into a solver. In order to use a primitive service transparently and take advantage of the solver classes already present in 2p-kt, we employed a client proxy. The client proxy wraps the actual client in the structure of a local `Primitive` interface, so that the connection aspects are handled in a hidden layer and the client can be used as a standard primitive.

The client proxy's implementation adheres to the details specified in 3.4.2. The core element of the component is the `ClientSession` interface, shown in Listing 4.17. It is defined as `StreamObserver` of the service's replies and it is used

Listing 4.16: Python implementation of `nt/1`

```python
class __NtPrimitive(DistributedElements.DistributedPrimitive):

  def solve(self, request: DistributedElements.DistributedRequest) ->
     Generator[DistributedElements.DistributedResponse, None, None]:
    arg0 = request.arguments[0]
    if(arg0.HasField("var")):
      n = 0
      while(True):
        substitutions = {}
        substitutions[arg0.var] = Utils.buildConstantArgumentMsg(n)
        yield request.replySuccess(substitutions = substitutions)
        n += 1
    elif(arg0.HasField("numeric")):
      yield request.replySuccess(hasNext = False)
    else:
      yield request.replyFail()
```

Listing 4.17: Interface of the `ClientSession`

```kotlin
interface ClientSession : StreamObserver<PrimitiveMsg> {

    val solutionsQueue: Iterator<Solve.Response>

    companion object {
        fun of(
            request: Solve.Request<ExecutionContext>,
            channelBuilder: ManagedChannelBuilder<*>
        ): ClientSession =
            ClientSessionImpl(request, channelBuilder)
    }
}
```

when connecting with the server. Moreover, it exposes the iterator `solutionQueue` where the solutions generated by the remote primitive are retrievable.

A `ClientSession` instance is composed of 3 additional methods, inherited from `StreamObserver`:

`onNext()` runs the handling process for every received message.

`onCompleted()` is the callback for when the connection is closed.

`onError()` is executed when an error has occurred during communication, such as a sudden shutdown of the connection.

The `onNext()` method is defined as in Listing 4.18. If the reply received contains a value, it is deserialized and added to the queue of solutions. Otherwise, if

Listing 4.18: Implementation of method `onNext()` in `ClientSession`

```kotlin
override fun onNext(value: PrimitiveMsg) {
    if (value.hasResponse()) {
        if (!value.response.solution.hasNext) {
            responseStream.onCompleted()
            this.onCompleted()
        }
        queue.add(value.response.deserialize(scope, request.context))
    } else if (value.hasRequest()) {
        val request = value.request
        if (request.hasSubSolve()) {
            responseStream.onNext(
                sessionSolver.solve(request.id, request.subSolve)
            )
        } else if (...) {
            ...
        }
    }
}
```

a sub-task is requested, the corresponding operation is executed and its result is dispatched.

The actual execution of these operations is delegated to the `SessionSolver`, which keeps track of the solver's execution context and any ongoing sub-tasks. For example, if a sub-solve request is received, the `solve()` method of `SessionSolver` is invoked, which runs as illustrated in Listing 4.19:

- It firstly checks if any computation with the message's ID already exists, creating a new iterator of solutions if not present.
- It then returns the next available value and serializes the result in a `SubResponseMsg`.

The `solutionQueue` exposed by `ClientSession` is an iterator of `Solve.Response` objects, as displayed in Listing 4.20. When a new element is requested with the `next()` method, it sends a `NextMsg` to the server and then it awaits a new value on the queue of solutions. If the connection is closed, the iterator will be marked as exhausted setting the `hasNext` variable to `false`.

Finally, the `PrimitiveClientFactory` object takes care of instantiating the client proxies. The method shown in Listing 4.21 takes the address and port of the server as arguments, and runs these instructions:

- It gets the remote primitive's signature with the `getSignature` RPC.

Listing 4.19: Implementation of the `solve()` method in `SessionSolver`

```kotlin
override fun solve(id: String, event: SubSolveRequest): SolverMsg {
    val query = event.query.deserialize()
    computations.putIfAbsent(id, sessionSolver.solve(query, event.timeout
       ).iterator())
    val solution: Solution = computations[id]!!.next()
    return buildSubSolveSolutionMsg(
        id,
        Solve.Response(solution),
        computations[id]!!.hasNext()
    )
}
```

Listing 4.20: Implementation of the `solutionQueue` in `ClientSession`

```kotlin
override val solutionsQueue: Iterator<Solve.Response> =
    object : Iterator<Solve.Response> {

        override fun hasNext(): Boolean = !closed

        override fun next(): Solve.Response {
            if (hasNext()) {
                responseStream.onNext(
                    SolverMsg.newBuilder().setNext(EmptyMsg.
                        getDefaultInstance()).build()
                )
                return queue.takeFirst()
            } else {
                throw NoSuchElementException()
            }
        }
    }
```

Listing 4.21: Implementation of the `connectToPrimitive()` method

```kotlin
fun connectToPrimitive(
    address: String = "localhost",
    port: Int = 8080
): Pair<Signature, Primitive> {
    val channelBuilder = ManagedChannelBuilder.forAddress(address, port)
        .usePlaintext()
    val channel = channelBuilder.build()
    val signature = GenericPrimitiveServiceGrpc.newFutureStub(channel)
        .getSignature(EmptyMsg.getDefaultInstance()).get()
    channel.shutdown()
    channel.awaitTermination(TERMINATION_TIMEOUT, TimeUnit.SECONDS)
    return signature.deserialize() to Primitive(primitive(channelBuilder)
        )
}
```

Listing 4.22: Wrapping of a `ClientSession` in a 2p-kt `Primitive`

```kotlin
private fun primitive(
    channelBuilder: ManagedChannelBuilder<*>
): (Solve.Request<ExecutionContext>) -> Sequence<Solve.Response> = {
    ClientSession.of(it, channelBuilder).solutionsQueue.asSequence()
}
```

- It returns the signature coupled with a client wrapped in a 2p-kt primitive, which is created with the code in Listing 4.22.

The pair returned by the method can thus be included in a library and later imported in a `Solver` with the instructions contained in Listing 4.23.

Listing 4.23: Import of a remote primitive in a `Solver`

```
logicProgramming {
    solver = Solver.prolog.mutableSolverWithDefaultBuiltins(
        otherLibraries = Runtime.of(
            Library.of(
                libraryName,
                mapOf(
                    PrimitiveClientFactory.connectToPrimitive(
                        primitivesHost, port)
                )
            )
        )
    )
}
```

Chapter 5

Validation

In this chapter, we will enlist the two validation methods used to verify the system quality.

5.1 Test Suite

During the implementation phase of the prototype, we followed the Test Driven Development model (TDD), allowing us to incrementally build a test suite. The resulting classes use the default library `kotlin.test` and include some ad-hoc primitive services that employ both the primary mechanisms of Prolog and the available sub-tasks in a `DistributedPrimitive`. These remote primitives are imported by a 2p-kt solver and queried in various test functions in order to verify their compatibility with the framework and the handling of predicate solutions, backtracking, and side effects. The coverage results of the suite are shown in figure 5.1.

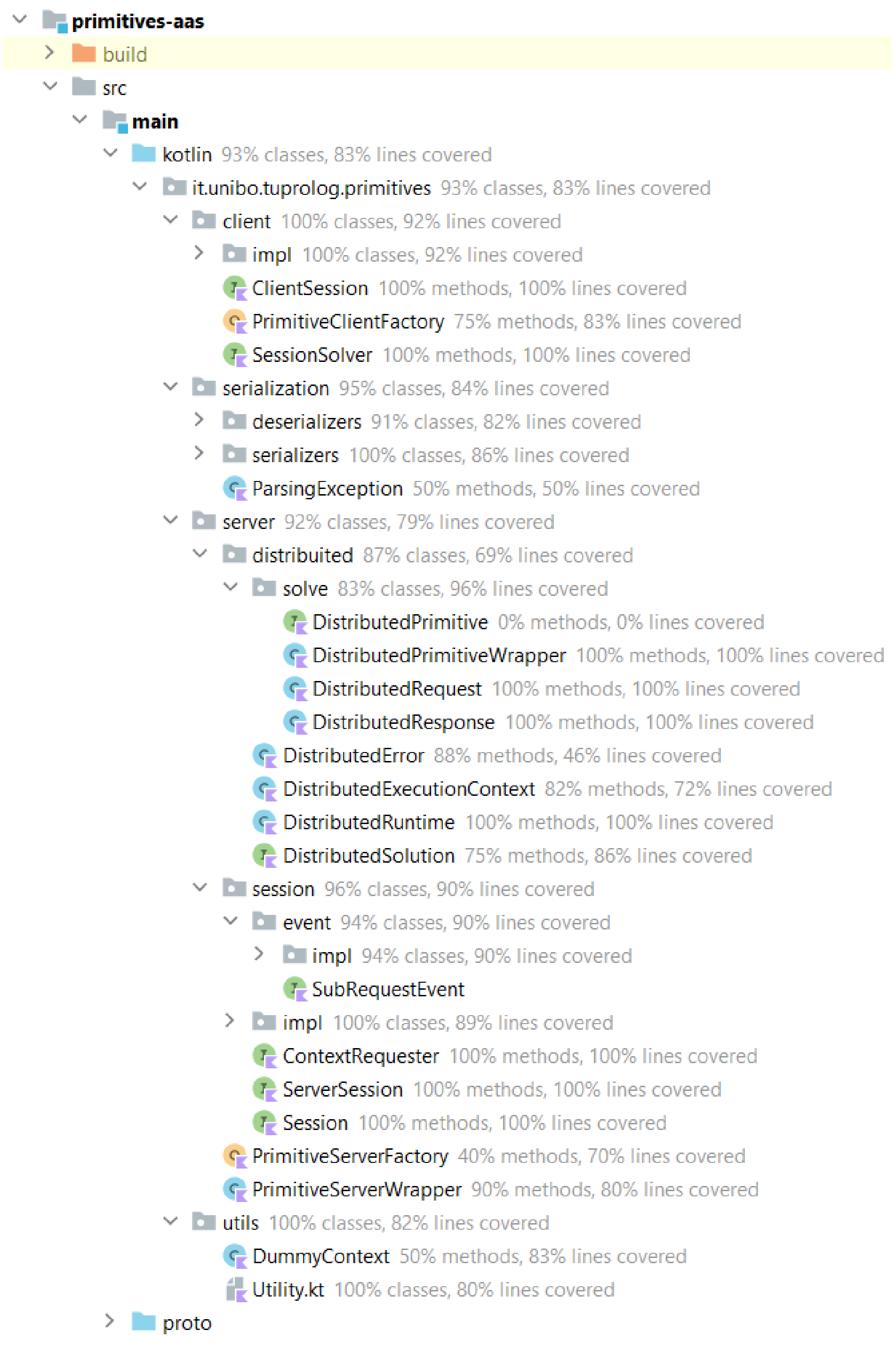
5.2 Real Application: The ml-lib

Finally, to verify the system's performance requirements and provide a concrete usage of the module, we decided to implement the library defined in [6]. This library includes methods to process large datasets, create machine learning predictors, and manage them in Prolog. The library is loaded on PyPI and can be installed with this command:

```
pip install prolog_primitives
```

5.2.1 Implementation

The Python project is composed of various nested packages:



```
primitives-aas
  build
  src
    main
      kotlin  93% classes, 83% lines covered
        it.unibo.tuprolog.primitives  93% classes, 83% lines covered
          client  100% classes, 92% lines covered
            impl  100% classes, 92% lines covered
            ClientSession  100% methods, 100% lines covered
            PrimitiveClientFactory  75% methods, 83% lines covered
            SessionSolver  100% methods, 100% lines covered
          serialization  95% classes, 84% lines covered
            deserializers  91% classes, 82% lines covered
            serializers  100% classes, 86% lines covered
            ParsingException  50% methods, 50% lines covered
          server  92% classes, 79% lines covered
            distribuited  87% classes, 69% lines covered
              solve  83% classes, 96% lines covered
                DistributedPrimitive  0% methods, 0% lines covered
                DistributedPrimitiveWrapper  100% methods, 100% lines covered
                DistributedRequest  100% methods, 100% lines covered
                DistributedResponse  100% methods, 100% lines covered
              DistributedError  88% methods, 46% lines covered
              DistributedExecutionContext  82% methods, 72% lines covered
              DistributedRuntime  100% methods, 100% lines covered
              DistributedSolution  75% methods, 86% lines covered
            session  96% classes, 90% lines covered
              event  94% classes, 90% lines covered
                impl  94% classes, 90% lines covered
                SubRequestEvent
              impl  100% classes, 89% lines covered
              ContextRequester  100% methods, 100% lines covered
              ServerSession  100% methods, 100% lines covered
              Session  100% methods, 100% lines covered
            PrimitiveServerFactory  40% methods, 70% lines covered
            PrimitiveServerWrapper  90% methods, 80% lines covered
          utils  100% classes, 82% lines covered
            DummyContext  50% methods, 83% lines covered
            Utility.kt  100% classes, 80% lines covered
      proto
```

Figure 5.1: Test suite's coverage results

`basic`: It contains the classes shown in 4.2.2 which implement the structure of a generic primitive service.

`generatedProto`: It is where the server's code generated from the `.proto` files is located.

`ml_lib`: It is the module containing all the 27 remote primitives' implementations, divided into nested packages depending on their functionality.

- `ml_lib.schema`: It handles the metadata of a dataset's domain.
- `ml_lib.dataset`: It manages the loading and reading of a dataset in clause form.
- `ml_lib.transformations`: It includes the primitives for the preprocessing of datasets.
- `ml_lib.predictors`: It presents the functions to create, fit and use a generic machine learning predictor.
- `ml_lib.neuralNetwork`: It contains the specific function to define a neural network, usable as a predictor.

When invoking a primitive, data can be passed as arguments either directly in `Term` form or through references. These references are identification strings generated by specific objects on the server side. These objects both collect and share data, enabling the exchange of information between the different remote primitives of the library. Each service listens on a specific port, from 8100 to 8126, and they are launched concurrently by an executor.

An example of ML primitive is `column/3`, which takes a reference for the dataset of interest and a column index as arguments and returns the values of that specific column in that dataset. The implementation of the primitive is shown in Listing 5.1. It runs the following instructions:

- Firstly it extracts the arguments from the requests and checks if they are the correct type.
- Using the reference provided by the client, it retrieves the corresponding dataset from `SharedCollection`, which is the central hub of data collections.
- It then gets the requested column, either by index or by name, and returns the serialized values in a `ResponseMsg`.
- If any of the previous steps fails, a failure is returned.

The library was also developed using TDD. We, in fact, extended the already existing test suite with specific functions that verified the correctness of the Python primitives.

Listing 5.1: Implementation of the remote primitive `column/3`

```python
class __ColumnPrimitive(DistributedElements.DistributedPrimitive):

    def solve(self, request: DistributedElements.DistributedRequest) ->
        Generator[DistributedElements.DistributedResponse, None, None]:
        dataset_ref = request.arguments[0]
        column = request.arguments[1]
        values = request.arguments[2]

        if(not dataset_ref.HasField("var") and values.HasField("var")):
            dataset = SharedCollections().getDataset(str(Utils.
                parseArgumentMsg(dataset_ref)))

            if(not column.HasField("var")):
                i = Utils.parseArgumentMsg(column)
                if(type(i) is not str):
                    i = dataset.column_names[int(i)]
                column = list(map(Utils.stringsConverter, tf.
                    get_static_value(dataset[i])))
                yield request.replySuccess({
                    values.var: Utils.fromListToArgumentMsg(column)
                }, hasNext=False)
            else:
                ...

        else:
            yield request.replyFail()

columnPrimitive = DistributedElements.DistributedPrimitiveWrapper("column
    ", 3, __ColumnPrimitive())
```

Listing 5.2: Implementation of the `preprocessing/3` clause

```
preprocessing(Dataset, Labels, Transformed) :-
    theory_to_schema(Schema),
    schema_transformation(Schema, A) ,
    normalize(A, Labels, B) ,
    fit(B, Dataset, C) ,
    transform(Dataset, C, Transformed).
```

Listing 5.3: Implementation of the `createModel/3` clause

```
createModel(NInput, NOutput, E) :-
  input_layer(NInput, A),
    dense_layer(A, 128, relu, B),
    dense_layer(B, 64, relu, C),
    output_layer(C, NOutput, linear, D),
    neural_network(D, E).
```

5.2.2 The Performance Comparison Test

A final test was devised in order to measure the system's impact on the performance. The demo consisted of building and fitting a Neural Network over the [20] dataset, implementing the code both in Python and in Prolog using `ml_lib`. In particular, the Prolog version used a `Solver` with the knowledge base composed of the facts that described the dataset and the following rules specific for machine learning operation:

`preprocessing/3` creates and fits on the dataset a preprocessing pipeline with a normalization step of the labels specified in the arguments.

`createModel/3` builds a neural network with two hidden layers and takes as arguments the number of neurons in the input and output layers.

`train_cv/3`, `train_cv_fold/4`, and `train_validate/4` are all used to execute cross-validation on the neural network previously created.

`test/3` verifies the quality of a predictor by calculating the mean average error of its predictions.

To run the demo in Prolog, the `Solver` is queried with the goal shown in Listing 5.6, which firstly loads the dataset on the server from theory form, then builds and applies the preprocessing pipeline and finally execute the cross-validation test.

In both languages, the code was run multiple times and the mean of all the execution times was calculated. As shown in Listing 5.7 and Listing 5.8, the

Listing 5.4: Implementation of the clauses for cross-validation

```prolog
train_cv(Dataset, LearnParams, AllPerformances) :-
  findall(Performance, train_cv_fold(Dataset, 5, LearnParams, Performance
      ), AllPerformances).

train_cv_fold(Dataset, K, LearnParams, Performance) :-
  fold(Dataset, K, Train, Validation),
  train_validate(Train, Validation, LearnParams, Performance).

train_validate(TrainingSet, ValidationSet, LearnParams, Performance) :-
  createModel(7, 1, NN),
  train(NN, TrainingSet, LearnParams, TrainedNN),
  test(NN, ValidationSet, Performance).
```

Listing 5.5: Implementation of the `test/3` clause

```prolog
test(NN, ValidationSet, Performance) :-
  predict(NN, ValidationSet, ActualPredictions),
  mae(ActualPredictions, ValidationSet, Performance).
```

Listing 5.6: Prolog query to define, fit and test the neural network

```prolog
theory_to_dataset(autoMpg, Dataset),
  preprocessing(Dataset, [cylinders, displacement, horsepower, weight,
      acceleration, 'model year', origin], Transformed),
  train_cv(Transformed, [max_epoch(100), loss(mse)], AllPerformances).
```

Listing 5.7: Output of demo executed in Python

```
...
10/10 [==============================] - 0s 9ms/step - loss: 6.2126 - mae
    : 1.8266 - val_loss: 24.4373 - val_mae: 3.2008
Epoch 99/100
10/10 [==============================] - 0s 6ms/step - loss: 6.0738 - mae
    : 1.8137 - val_loss: 23.2056 - val_mae: 3.1173
Epoch 100/100
10/10 [==============================] - 0s 6ms/step - loss: 6.1199 - mae
    : 1.8058 - val_loss: 23.5669 - val_mae: 3.1574
3/3 [==============================] - 0s 4ms/step
Mean Absolute Error: 2.050995111465454
Medium time calculated: 25.01106333732605
```

performances are similar, and the distributed aspects of the 2p-kt system don't affect the performance out-of-reasonable bounds.

Listing 5.8: Output of demo executed in 2p-kt

```
 1/10 [==>...........................] - ETA: 0s - loss: 9.4589 - mae:
    2.3689
10/10 [==============================] - 0s 8ms/step - loss: 6.4137 - mae
    : 1.8330 - val_loss: 9.7842 - val_mae: 2.3726
Epoch 98/100
 1/10 [==>...........................] - ETA: 0s - loss: 9.6123 - mae:
    2.1338
10/10 [==============================] - 0s 6ms/step - loss: 6.4160 - mae
    : 1.8314 - val_loss: 10.7024 - val_mae: 2.4423
Epoch 99/100
 1/10 [==>...........................] - ETA: 0s - loss: 7.3944 - mae:
    1.9367
10/10 [==============================] - 0s 6ms/step - loss: 6.3663 - mae
    : 1.8305 - val_loss: 9.8800 - val_mae: 2.4473
Epoch 100/100
 1/10 [==>...........................] - ETA: 0s - loss: 6.9109 - mae:
    1.7050
10/10 [==============================] - 0s 8ms/step - loss: 6.3447 - mae
    : 1.8339 - val_loss: 9.8858 - val_mae: 2.4383
1/3 [=========>....................] - ETA: 0s
3/3 [==============================] - 0s 1ms/step
Yes(query=((getDataset(Dataset_66), preprocessing(Dataset_66, [cylinders,
    displacement, horsepower, weight, acceleration, 'model year', origin
    ], Transformed_28)), train_cv(Transformed_28, [max_epoch(100), loss(
    mse)], AllPerformances_28)), substitution={Dataset_66=qzhxcxtiaf,
    Transformed_28=kaxlyzmhph, AllPerformances_28=[2.5825018882751465,
    2.0743658542633057, 1.8034321069717407, 2.304511785507202,
    2.2109456062316895]})
Average time 25.509640000000008
```

Chapter 6

Conclusions

In this thesis, we explore the possibility of creating a bridge to connect the logic realm, in particular the Prolog language, with external multi-platform entities.

The current state of the art for the logic paradigm emphasizes how difficult it is to incorporate symbolic AI with other topics of research. To solve this problem, we propose a solution that exploits the concept of primitives and the As-A-Service model. More specifically, our system consists of a client-server structure where the solver acts as a client and the primitive is an external service used to generate values.

To demonstrate the feasibility of our approach, we design a generic service interface, the message protocol, and the characteristics of the distributed entities. Furthermore, we illustrate the system functionalities by providing a prototype that extends the 2p-kt framework and employs the Interface Description Languages gRPC and protobuf. Thanks to these technologies, multiple languages can be used to develop primitive services, as it is displayed in the prototype's code.

Finally, we present an example of the system's usage by implementing the library described in [6], which is composed of Prolog predicates for machine learning operations. A demo that uses the library has, on average, a similar execution time to a functionally equivalent program in pure Python, which proves that the employment of remote primitives does not cause a significant deterioration of performance.

From our perspective, the results this thesis accomplishes open the door to a vast amount of new possibilities for symbolic AI. In fact, the system enables users to develop their custom primitives and libraries for Prolog without worrying about the restrictions imposed by logic programming.

In other words, it is feasible to define methods that exploit functionalities from the most diverse platforms and paradigms without leaving the logic realm. It would be interesting, for example, to develop hybrid systems that combine deep learning and automated reasoning, such as the ones theorized in [14]. These technologies could acquire knowledge about objects' representations as well as their relations with minimal prior understanding of the data structure. Achieving the ability to learn from the environment and, at the same time, reason the results would be a significant milestone in the field of AI.

It is discussed in [3] that the combination of machine learning's "black-box" techniques with symbolic AI's "white-box" methodologies would generate partially understandable approaches and improve the scalability, reactivity, and accountability of any intelligent system in the IoT context. Data science would also benefit from the intersection with symbolic AI since it would open up new research directions with the aim of building knowledge-based, automated methods for scientific discovery [13].

In conclusion, we expect that, in the future of technological research, our approach will be utilized to create hybrid solutions that for now are only theoretical.

Bibliography

[1] Krzysztof R. APT. Chapter 10 - logic programming. In JAN VAN LEEUWEN, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 493–574. Elsevier, Amsterdam, 1990.

[2] Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[3] Roberta Calegari, Giovanni Ciatto, Stefano Mariani, Enrico Denti, and Andrea Omicini. Lpaas as micro-intelligence: Enhancing iot with symbolic reasoning. *Big Data and Cognitive Computing*, 2(3):23, 2018.

[4] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. 2p-kt: A logic-based ecosystem for symbolic ai. *SoftwareX*, 16:100817, 2021.

[5] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. Lazy stream manipulation in prolog via backtracking: The case of 2p-kt. In *Logics in Artificial Intelligence: 17th European Conference, JELIA 2021, Virtual Event, May 17–20, 2021, Proceedings*, pages 407–420. Springer, 2021.

[6] Giovanni Ciatto, Matteo Castigliò, and Roberta Calegari. Logic programming library for machine learning: API design and prototype. In Roberta Calegari, Giovanni Ciatto, and Andrea Omicini, editors, *CILC 2022 – Italian Conference on Computational Logic*, volume 3204 of *ceurws*, pages 104–118. CEUR-WS, 2022.

[7] Chris Currier. Protocol buffers. In *Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*, pages 223–260. Springer, 2022.

[8] docs.python.org. Python documentation. `https://docs.python.org/3/`, 2012. accessed on 02-September-2023.

[9] Yucong Duan, Guohua Fu, Nianjun Zhou, Xiaobing Sun, Nanjangud C Narendra, and Bo Hu. Everything as a service (xaas) on the cloud: origins, current

and future trends. In *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 2015.

[10] Werner Frey and Uwe Reyle. A prolog implementation of lexical functional grammar as a base for a natural language processing system. In *First Conference of the European Chapter of the Association for Computational Linguistics*, 1983.

[11] Marta Garnelo and Murray Shanahan. Reconciling deep learning with symbolic artificial intelligence: representing objects and relations. *Current Opinion in Behavioral Sciences*, 29:17–23, 2019.

[12] Ben Goertzel. Perception processing for general intelligence: Bridging the symbolic/subsymbolic gap. In *International Conference on Artificial General Intelligence*, pages 79–88. Springer, 2012.

[13] Robert Hoehndorf, Núria Queralt-Rosinach, et al. Data science and symbolic ai: Synergies, challenges and opportunities. *Data Science*, 1(1-2):27–38, 2017.

[14] Eleni Ilkou and Maria Koutraki. Symbolic vs sub-symbolic ai methods: Friends or enemies? In *CIKM (Workshops)*, volume 2699, 2020.

[15] Matthias Jarke, Jim Clifford, and Yannis Vassiliou. An optimizing prolog front-end to a relational query system. *ACM SIGMOD Record*, 14(2):296–306, 1984.

[16] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V Hermenegildo, Jose F Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, et al. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, 22(6):776–858, 2022.

[17] Ewing L Lusk and Ross A Overbeek. The automated reasoning system itp. Technical report, CM-P00069210, 1984.

[18] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[19] Bernard A Nadel. Constraint satisfaction in prolog: Complexity and theory-based heuristics. *Information sciences*, 83(3-4):113–131, 1995.

[20] R. Quinlan. Auto MPG. UCI Machine Learning Repository, 1993. DOI: https://doi.org/10.24432/C5859H.

[21] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[22] Richard Snodgrass. *The interface description language: definition and use.* Computer Science Press, Inc., 1989.

[23] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.

[24] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3):75–86, 1993.