Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Department of Computer Science

Software Engineering Orientation

Semester 6 project

2024

Prolog remote constraint logic programming

Technical documentation

Noah Godel

Superviseurs: Frédéric Bapst

Fribourg, 11 March 2024

Version 1.0

Hes·so

Table of versions

Version	Date	Modifications
1.0	11.3.2024	First version

Contents

Section 1
Introduction

This report presents the activities carried out during the semester 6 project of the Software Engineering Bachelor programme at the School of Engineering and Architecture of Fribourg. The next section provides an overview of the project. For further details, please refer to the specification. The project repository is located at the following URL.

https://gitlab.forge.hefr.ch/frederic.bapst/24-ps6-remote-clp

1.1. Context

The purpose of this project is to develop a solution that bridges the gap between Prolog and Operations Research (OR) libraries, such as Google OR Tools and Gecode. The goal is to create a web-based system that allows multiple users to access and utilize these OR libraries for Constraint Logic Programming (CLP) in Prolog.

Previous attempts to integrate Prolog with OR libraries, such as Google OR Tools, required local installation of the libraries, limiting accessibility. This project aims to overcome these limitations by developing a web-based API that eliminates the need for local installations. The proposed architecture leverages the scalability and resilience of a Kubernetes cluster to host the web API, making it well-suited for deployment in various operational scenarios.

1.2. Goal

The primary goals of this project are:

- Implement a web API that provides access to the CP-SAT solver of the Google OR Tools library for CLP in Prolog. The API should be designed to handle multiple concurrent requests and provide scalability for increased usage in the future. A token-based authentication mechanism will be implemented to ensure secure access to the API.
- Deploy the web API on a Kubernetes cluster to ensure scalability and resilience, with automated deployment using Gitlab CI/CD pipelines.
- Develop an OS-independent client library for SWI-Prolog that allows easy access to the web API, in a way similar to other CLP libraries in Prolog.
- Perform a series of tests, including unit tests and performance tests, to ensure the reliability and performance of the web API.
- Implement a series of demonstration programs to showcase the capabilities of the web API and the client library.

1.3. Document structure

This document is structured as follows.

- Introduction: this section presents the context, the goal and the structure of the document.
- Analysis: this section presents the constraints, exploration of different technologies and features and the technological choices.
- Design: this section presents the design of the Prolog client library and the remote CLP service.
- Implementation: this section presents the implementation of the Prolog client library and the remote CLP service.
- Results: this section presents the challenges and the future work.
- Conclusion: this section concludes the document.

Section 2
Analysis

This section presents the constraints, the exploration of different technologies and features and the technological choices.

2.1. Constraints

An important constraint this project has is that the client has to work cross platform. Since SWI-Prolog uses very little OS specific features this will not be hard to accieve. But it is important to check that the client library works on all OS in the same manner.

This project additionally has the following constraints.

2.1.1. Authentication

To insure the security of the service, the client must authenticate itself. Since a constraint logic programming service can be quite compute intensive, the service must be able to identify the client and limit the number of requests.

To implement this, we will use a token based authentication system. The client will request a token from the service and use it in all subsequent requests. The rate of requests will be limited by the token.

To manage the tokens an administration interface need to be implemented. This interface will allow the administrator to create, delete and list the tokens.

2.1.2. Concurrency

As already mentioned, the service can be quite compute intensive. One of the constraints of this project is to be able to handle multiple requests concurrently.

It is also specified that the service will be deployed on the institution's kubernetes cluster. This means that the service must be able to scale horizontally. A simple way to achieve this is to use a message queue to distribute the requests to multiple workers.

To ensure that the message queue doesn't overfill, the rate limiting system will be implemented at the API Gateway level.

2.1.3. Prolog library usage

The client library must feel similar to use as other CLP libraries that are available in Prolog. The structure of the library is inspired by the clpfd library. The usage of the

API must be hidden from the user as much as possible, they only need to specify the token and the host.

Here is an example of how CLP programs looks.

TODO add example CLP program

2.2. How Google OR-Tools MathOpt API works

MathOpt is a library within Google OR-Tools that provides a set of tools to solve mathematical optimization problems. Like linear programming and mixed-integer programming. It makes abstraction of the solver used and provides a common interface to solve the problems.

MathOpt provides a Web API that allows to solve optimization problems remotely on Google's servers. The API is a REST API that uses JSON to encode the optimization problem and the solution.

The usage of the remote API is quite simple. The user creates a problem, sets the variables and constraints as usual. To send it to the remote service, the user must call a remote solve function that takes an API key and the model object. The function returns the solution object.

The model object is parsed to JSON so it can be sent to the API. Listing 1 python code shows how this works.

```python
model = mathopt.Model(name="my_model")
x = model.add_binary_variable(name="x")
y = model.add_variable(lb=0.0, ub=2.5, name="y")
model.add_linear_constraint(x + y <= 1.5, name="c")
model.add_linear_constraint(2*x + y >= -13, name="d")
model.maximize(2 * x + y)
result, logs = remote_http_solve.remote_http_solve(
    model, mathopt.SolverType.GSCIP, api_key=api_key
)
```

Listing 1: Python code to create a MathOpt model

The code above creates a model with two variables, x and y, and two constraints, c and d. The model is then sent to the remote service and the solution is returned. Listing 2 shows the JSON representation of the model.

```
{
    "solverType": "SOLVER_TYPE_GSCIP",
    "model": {
        "name": "my_model",
        "variables": {
            "ids": [ "0", "1" ],
            "lowerBounds": [ 0.0, 0.0 ],
            "upperBounds": [ 1.0, 2.5 ],
            "integers": [ true, false ],
            "names": [ "x", "y" ]
        },
        "objective": {
            "maximize": true,
            "linearCoefficients": {
                "ids": [ "0", "1" ],
                "values": [ 2.0, 1.0 ]
            }
        },
        "linearConstraints": {
            "ids": [ "0", "1" ],
            "lowerBounds": [ "-Infinity", -13.0 ],
            "upperBounds": [ 1.5, "Infinity" ],
            "names": [ "c", "d" ]
        },
        "linearConstraintMatrix": {
            "rowIds": [ "0", "0", "1", "1" ],
            "columnIds": [ "0", "1", "0", "1" ],
            "coefficients": [ 1.0, 1.0, 2.0, 1.0 ]
        }
    }
}
```

Listing 2: JSON representation of a MathOpt model

2.3. Long running requests in Prolog

Since the solving of constraint logic programming problems can be quite compute intensive, the service must be able to handle long running requests from the client. The service must be able to handle requests that take several minutes to solve.

Handleing this with a simple blocking request would not be a good idea. The connection could be lost or the client could timeout. To handle this, the service must be able to handle long running requests in a non-blocking way.

The RESTful way to handle this is to return a 202 Accepted status code with a location header that points to a status endpoint. The client can then poll this endpoint to get the status of the request. When the request is finished, the status endpoint will return a 200 OK status code with the location of the result. Figure 1 below shows how this works.

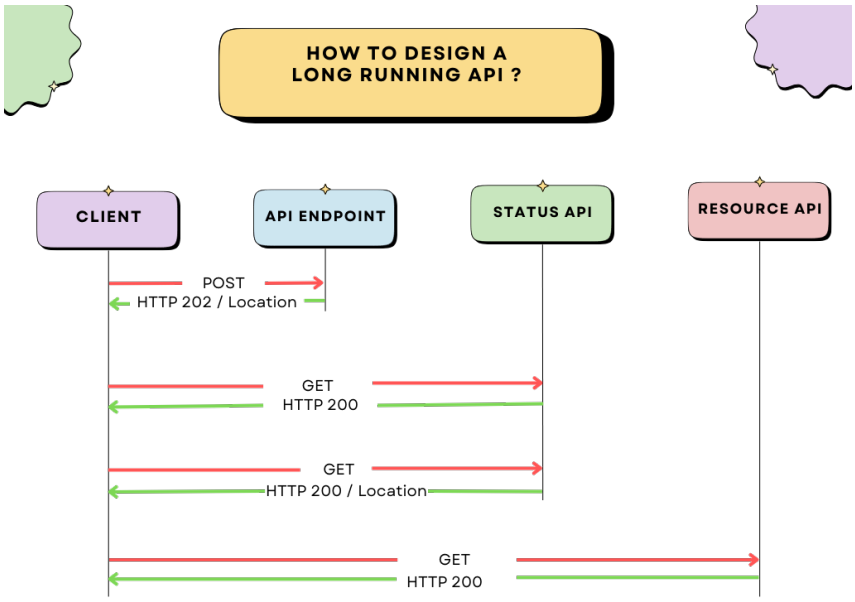

Figure 1: Long running requests in a RESTful API

On the Prolog side, the client library must be able to handle this. The client must be able to send a request and poll the status endpoint until the request is finished. When the request is finished, the client must be able to get the result.

SWI-Prolog has built-in libraries to handle HTTP requests, JSON parsing and serialization and threading. This makes it quite easy to implement this in Prolog. Listing 3 shows a Node.js server that simulates a long running request.

```
var http = require('http');

var value = 0;

http.createServer((req, res) => {
    if (req.url === '/solve') {
        setTimeout(() => { value = 42; }, 5000);
        res.writeHead(202);
        res.end();
        return;
    } else if (req.url === '/status') {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify(value > 0));
        return;
    } else if (req.url === '/value') {
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify(value));
        return;
    }
}).listen(3000);
```

Listing 3: Node.js server that simulates a long running request

A Prolog client that polls the status endpoint until the request is finished.

```
:- use_module(library(http/http_open)).
:- use_module(library(http/json)).

check :-
    http_open('http://localhost:3000/status', In, []),
    json_read(In, Json, [true(t), false(f)]),
    close(In),
    get_time(T),
    write('status: '), write(Json), write(' '), writeln(T),
    Json = t.

polling :-
    write('Starting'), nl,
    http_open('http://localhost:3000/solve', In, []),
    close(In),
    % poll the status endpoint until the request is finished
    thread_wait(check, [retry_every(1)]),
    write('Fetching result'), nl,
    http_open('http://localhost:3000/value', In2, []),
    json_read(In2, Json),
    close(In2),
    writeln(Json).
```

Listing 4: Prolog client that polls the status endpoint

The output of the Prolog client is shown in Listing 5.

```
?- polling.
Starting
status: f 1712614154.6943989
status: f 1712614155.695983
status: f 1712614156.6980333
status: f 1712614157.700015
status: f 1712614158.7022102
status: t 1712614159.7041478
Fetching result
42
true.
```

Listing 5: Output of the Prolog client

2.4. Metadata in Prolog variables

To be able to send the variables and constraints to the service, the client library must be able to keep track of metadata in the variables. This metadata is used to identify the variables and the constraints in the service.

Listing 6 is an example of how this could be implemented in SWI-Prolog.

```
% hook that is called when a variable is unified
attr_unify_hook(M, E) :-
    writeln('Variable was unified with:'),
    write('value: '), writeln(E),
    write('had rclp attribute: '), writeln(M).

define([], _).
define([Var|Ls], N) :-
    % add the rclp attribute to the variable with value N
    put_attr(Var, rclp, N),
    N1 is N + 1,
    define(Ls, N1).

define(Ls) :-
    define(Ls, 1).

solve([]).
solve([Var|Ls]) :-
    % get the rclp attribute from the variable
    get_attr(Var, rclp, N),
    writeln(N),
    solve(Ls).

go :-
    define([X,Y,Z]),
    solve([X,Y]),
    Z = a.
```

Listing 6: Prolog variables with metadata

The output of the Prolog program is shown in Listing 7.

```
?- go.
1
2
Variable was unified with:
value: a
had rclp attribute: 3
true.
```

Listing 7: Output of the Prolog program

2.5. Technological choices

In this section, we will present the technological choices that have been made for this project.

2.5.1. Prolog engine

The Prolog engine that will be used for this project is SWI-Prolog. SWI-Prolog is a free implementation of Prolog that is widely used in the industry. It has a large set of libraries that make it easy to implement the client library.

In the case of the client for the remote CLP service, SWI-Prolog is also a good choice. It has built-in libraries to handle HTTP requests, JSON parsing and serialization and threading. This makes it easy to implement the client.

However, in the fututre there is interest in using GNU Prolog since it is used in the institution. This would require a complete rewrite of the client library. From an initial analysis, it seems that GNU Prolog doesn't have built-in libraries to handle HTTP requests directly. This would require to use a C library to handle the requests or to directly use the TCP sockets.

2.5.2. Web Service

The API Gateway is the entry point of the service. It is responsible for authenticating the client and rate limiting the requests. It finally forwards the requests to the message queue.

For the API Gateway, a simple but fast language is needed. **Go** is a good choice for this. It is fast, has a good standard library and is easy to deploy. RESTful APIs are well supported in Go.

The message queue is responsible for distributing the requests to the workers. **Rabbit MQ** is a good choice for this. It is a robust message queue that is easy to deploy and has good support for multiple languages.

The workers are responsible for solving the requests. In our case we need to choose a language that is supported by Google OR-Tools. **Python** is a good choice for this. It is well supported by Google OR-Tools and is easy to use and deploy.

Section 3
Design

This section presents the design of the Prolog client library and the remote CLP service.

3.1. Prolog client library

The Prolog client library is a library that allows the user to access the remote CLP service from Prolog. The library should be easy to use and feel similar to other CLP libraries that are available in Prolog.

3.1.1. Structure

3.1.2. Usage example

Listing 8 is an example of how the library should be used.

```
:- include('remote-clp.pl').

pyth_triplets(N,Ls1) :-
    Ls1 = [A1,B1,C],
    Ls = [U,V,K, A,B,C],
    rclp_fd_domain(Ls,1,N),
    A*A + B*B ~#= C*C,
    UU ~#= U*U,
    VV ~#= V*V,
    (U-V) rem 2 ~#= 1,
    coprime(U,V, Aux),
    A ~#= K*(UU - VV),
    C ~#= K*(UU + VV),
    B ~#= K*(2*U*V),
    append([UU, VV|Ls], Aux, AllVariables),
    rclp_fd_labeling(AllVariables),
    unbreak_symmetry(A,B, A1,B1).

coprime(A,B, [X,Y]) :-
    X~#=<B, Y~#=<A,
    (A*X - B*Y ~#= 1).

unbreak_symmetry(A,B, A,B).
unbreak_symmetry(A,B, B,A).

optimizeDemo(A,B) :-
    rclp_fd_domain([A,B],0, 50),
    Z ~#= 3*A + 2*B,
    A+B ~#< 50,
    4*A-B ~#< 88,
    rclp_fd_maximize(rclp_fd_labeling([A,B]), Z).
```

Listing 8: Example of a Prolog program using the client library

3.2. Remote CLP Service

The remote CLP service is a web service that provides access to the CP-SAT solver of the Google OR Tools library for CLP in Prolog. The service should be able to handle multiple concurrent requests and provide scalability for increased usage in the future.

3.2.1. Kubernetes architecture

TODO add architecture diagram

3.2.2. Client-server communication

TODO sequence diagram

3.2.3. Request data structure

3.2.4. Result data structure

3.3. Testing strategy

Section 4
Implementation

TODO

Section 5

Results

Section 6
Conclusion

6.1. Challenges

6.2. Future work

6.3. Personal opinion

TODO

Declaration of honor

In this project, we used generative AI tools, namely Github Copilot for coding and TODO for paraphrasing. Copilot was employed as an advanced autocomplete feature, but it did not generate a significant portion of the project. Writefull, on the other hand, was utilised to enhance the clarity of this document by employing its paraphrasing and grammar checking capabilities. I, the undersigned Noah Godel, solemnly declare that the submitted work is the result of personal effort. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and author citations have been clearly acknowledged.

Bibliography

Table of figures