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, 17 May 2024

Version 1.0

Hes·so

Table of versions

Version	Date	Modifications
1.0	17.5.2024	Final version
0.2	15.5.2024	Corrected some typos, added references and add design diagrams
0.1	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 program 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 [11]. 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 [8] 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

This project has several constraints that need to be taken into account during the development process. One of them is that the client library must work cross-platform. SWI-Prolog can be installed on all the most popular operating systems (Windows, macOS, Linux) and the client library should work in the same way on all of them. This means that no platform specific code should be used in the client library.

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 user (the administrator to be specific) 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 could be used at the job dispatcher 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 SWI-Prolog [10]. 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.

Listing Listing 1 (inspired by [4]) is an example of how a CLP programs should look like.

```
:- use_module(library(clpfd)).

alpha(Ls) :-
    Ls = [A, B, C, _D, E, F, G, H, I, J, K, L, M,
          N, O, P, Q, R, S, T, U, V, W, X, Y, Z],
    Ls ins 1..26,
    all_distinct(Ls),
    B + A + L + L + E + T #= 45,
    C + E + L + L + O #= 43,
    C + O + N + C + E + R + T #= 74,
    F + L + U + T + E #= 30,
    F + U + G + U + E #= 50,
    G + L + E + E #= 66,
    J + A + Z + Z #= 58,
    L + Y + R + E #= 47,
    O + B + O + E #= 53,
    O + P + E + R + A #= 65,
    P + O + L + K + A #= 59,
    Q + U + A + R + T + E + T #= 50,
    S + A + X + O + P + H + O + N + E #= 134,
    S + C + A + L + E #= 51,
    S + O + L + O #= 37,
    S + O + N + G #= 61,
    S + O + P + R + A + N + O #= 82,
    T + H + E + M + E #= 72,
    V + I + O + L + I + N #= 100,
    W + A + L + T + Z #= 34,
    label(Ls).
```

Listing 1: Example of a CLP program

2.2. How Google OR-Tools MathOpt API works

MathOpt [6] 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 formatted in JSON so it can be sent to the API. Listing 2 Python code shows how this works.

```
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 2: 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 3 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 3: 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.

Handling 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.

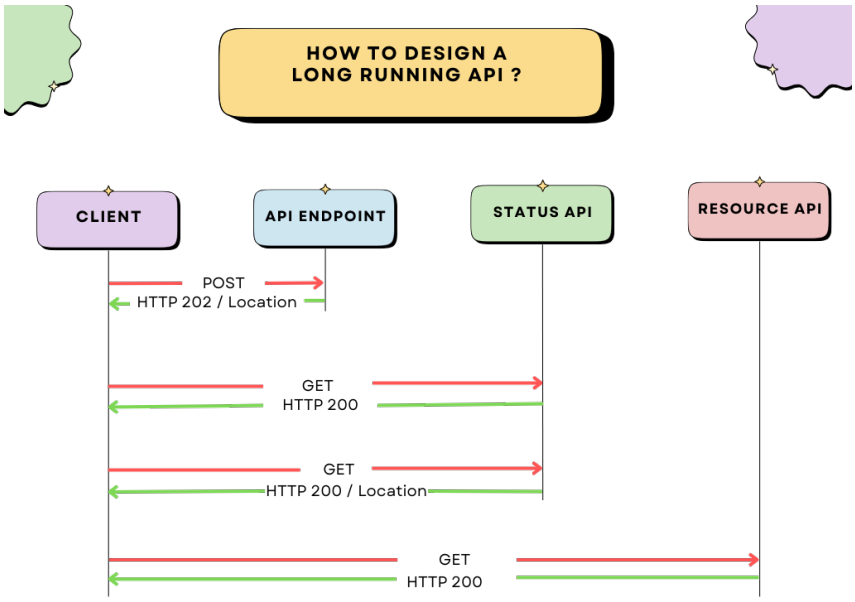

Figure 1: Long running requests in a RESTful API

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 4 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 4: 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 :-
    writeln('Starting'),
    http_open('http://localhost:3000/solve', In, []),
    close(In),
    % poll the status endpoint until the request is finished
    thread_wait(check, [retry_every(1)]),
    writeln('Fetching result'),
    http_open('http://localhost:3000/value', In2, []),
    json_read(In2, Json),
    close(In2),
    writeln(Json).
```

Listing 5: Prolog client that polls the status endpoint

The output of the Prolog client is shown in Listing 6.

```
?- 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 6: 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 7 is an example of how this could be implemented in SWI-Prolog.

```
% hook (callback) that is called when a variable is unified
% these hooks are bound to the module so they don't interfere with other
attributes
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 7: Prolog variables with metadata

The output of the Prolog program is shown in Listing 8.

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

Listing 8: 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 future 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 job dispatcher is the entry point of the service. It is responsible for authenticating the client and rate limiting (if implemented) the requests. It finally forwards the requests to the message queue.

For the job dispatcher, a simple but fast language is needed. **Go** [7] with **Fiber** [9] 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** [3] 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** [12] is a good choice for this. It is well supported by Google OR-Tools and is easy to use and deploy.

2.5.3. Authentication

For the authentication system, a simple JWT token [2] based system will be used. This is a simple and secure way to authenticate the client. One of the advantages of JWT is that they don't need to be stored on the server. This makes it easy to scale the service. For now there will be no persistence of the tokens. This could be added in the future. Token expiration will be implemented to ensure that the tokens are not valid forever.

To implement a potential token revocation system, a simple database could be used. It is simple to use and doesn't require a server.

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 of the client library

- The `remote_clp` module is the main module of the client library. It contains the predicates that the user will use to interact with the remote service. To use the client library, the user must first load this module with the `use_module/1` predicate.
- `api_config/1`: predicate that configures the client library with the URL of the remote service and the token.
- All the other predicates are used to specify the constraints and the variables of the problem (the operators and predicates are identical on the CLPFD library [10]. The user can use these predicates to specify the problem and then call `label/1` (without options) or `labeling/2` (with options) with the list of variables to solve the problem.

Here is a exhaustive list of the predicates that the user can use:

- `fd_var/1`: specifies that the variable is a finite domain variable.
- `api_config/1`: specifies the URL of the remote service and the token, this predicate must be called before any other predicate and takes a list of compound terms as argument.
  - `url/1`: specifies the URL of the remote service.
  - `key/1`: specifies the token.
- `in/2`: specifies the domain of a variable, the domain is specified as a range (e.g., 1..10 1 and 10 included).
- `ins/2`: same as `in/2` but for a list of variables.
- `label/1`: solves the problem with the default options.
  - The argument is a list of variables.
  - All the constraints that involve theses variables are included in the problem (and all the variables that are involved in the constraints also are implicitly included).
- `labeling/2`: solves the problem with the specified options, the options are specified as a list of compound terms and only one objective (min or max) can be specified. If there is an objective, the solver will return only one solution.

  - The first argument is a list of variables (same as `label/1`).
  - `min/1`: specifies the variable to minimize.
  - `max/1`: specifies the variable to maximize.
  - `time_limit/4`: specifies the time limit of the solver in hours, minutes, seconds and milliseconds (e.g., 1, 1, 1, 1 for 1 hour, 1 minute, 1 second and 1 millisecond).
  - `solution_limit/1`: specifies the number of solutions to find.
- `all_different/1`: specifies that all the variables must have different values.
- Arithmetic constraints: `#>`, `#<`, `#=`, `#>=`, `#=<` and `#\=` are supported and can take any arithmetic expression (operators `+`, `-`, `*`, `//` and `mod` are supported) as arguments.

Listings 9, 10, 11 and 12 are example programs that show how the client library should be used. These examples will also be used to test the client library.

3.1.2. Example programs

Listing 9 shows the N-queens problem in Prolog. The program finds a solution to the N-queens problem for a given N. The program uses the `n_queens/2` predicate to find the solution. This program illustrates how to work with lists of variables through the usage of the `all_different/1` and `ins/2` constraints.

```
:- use_module('../remote_clp').
:- api_config([url('https://remote-clp.kube.isc.heia-fr.ch/api'),
key('<jwt_token>')]).

n_queens(N, Qs) :-
    length(Qs, N),
    Qs ins 1..N,
    all_different(Qs),
    diag(Qs, Ds1, Ds2),
    all_different(Ds1),
    all_different(Ds2),
    append(Qs, Ds1, Vs1),
    append(Vs1, Ds2, Vs),
    label(Vs).

diag([], [], []).
diag(Qs, Ds1, Ds2) :-
    diag(Qs, Ds1, Ds2, 0).

diag([], [], [], _).
diag([Q|Qs], [D1|Ds1], [D2|Ds2], N) :-
    D1 #= Q + N,
    D2 #= Q - N,
    N1 is N + 1,
    diag(Qs, Ds1, Ds2, N1).
```

Listing 9: N-queens problem in Prolog

Listing 10 shows the Pythagorean triplets problem in Prolog. The program finds all Pythagorean triplets with integers below N. The program uses the `pyth_triplets/2` predicate to find the solution. This program illustrates the usage of non-linear constraints (in this case $a^2 + b^2 = c^2$) and the idea of symmetry breaking.

```
:- use_module('../remote_clp').
:- api_config(...).

pyth_triplets(N,Ls1) :-
    Ls1 = [A1,B1,C],
    Ls = [A,B,C],
    Ls ins 1..N,
    A #=< B, B #=< C,
    A*A + B*B #= C*C,
    label(Ls),
    unbreak_symmetry(A,B,A1,B1).

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

Listing 10: Pythagorean triplets with integers below N in Prolog

Listing 11 shows a simple optimization problem in Prolog. The program finds the values of A and B that maximize the expression $z = 3a + 2b$, subject to the constraints $a + b < 50$ and $4a - b < 88$. The program uses the `optimizeDemo/1` predicate to find the solution. This program illustrates how to work with optimization problems.

```
:- use_module('../remote_clp').
:- api_config(...).

optimizeDemo(Ls) :-
    Ls = [A,B],
    Ls ins 0..50,
    Z #= 3*A+2*B,
    A+B #< 50,
    4*A-B #< 88,
    labeling([max(Z)], [A,B]).
```

Listing 11: Simple optimization problem in Prolog

Listing 12 shows a word puzzle in Prolog. The numbers 1 - 26 have been randomly assigned to the letters of the alphabet. The numbers beside each word are the total of the values assigned to the letters in the word. e.g for LYRE L,Y,R,E might equal 5,9,20 and 13 respectively or any other combination that add up to 47. The program uses the `solve_puzzle/1` predicate to find the solution. This program illustrates how to work with linear constraints.

```
:- use_module('../remote_clp').
:- api_config(...).

solve_puzzle(Ls) :-
    Ls = [A, B, C, _D, E, F, G, H, I, J, K, L, M,
          N, O, P, Q, R, S, T, U, V, W, X, Y, Z],
    Ls ins 1..26,
    all_different(Ls),
    B + A + L + L + E + T #= 45,
    C + E + L + L + O #= 43,
    C + O + N + C + E + R + T #= 74,
    F + L + U + T + E #= 30,
    F + U + G + U + E #= 50,
    G + L + E + E #= 66,
    J + A + Z + Z #= 58,
    L + Y + R + E #= 47,
    O + B + O + E #= 53,
    O + P + E + R + A #= 65,
    P + O + L + K + A #= 59,
    Q + U + A + R + T + E + T #= 50,
    S + A + X + O + P + H + O + N + E #= 134,
    S + C + A + L + E #= 51,
    S + O + L + O #= 37,
    S + O + N + G #= 61,
    S + O + P + R + A + N + O #= 82,
    T + H + E + M + E #= 72,
    V + I + O + L + I + N #= 100,
    W + A + L + T + Z #= 34,
    label(Ls).
```

Listing 12: Word puzzle in Prolog

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 deployment architecture

Figure 2 shows the deployment architecture of the remote CLP service. The service is deployed on a Kubernetes cluster.

The job dispatcher is exposed to the internet through the ingress and is responsible for authenticating the client and sending the requests to the message queue. The message queue distributes the requests to the workers. The workers solve the requests and send the results back to the job dispatcher.

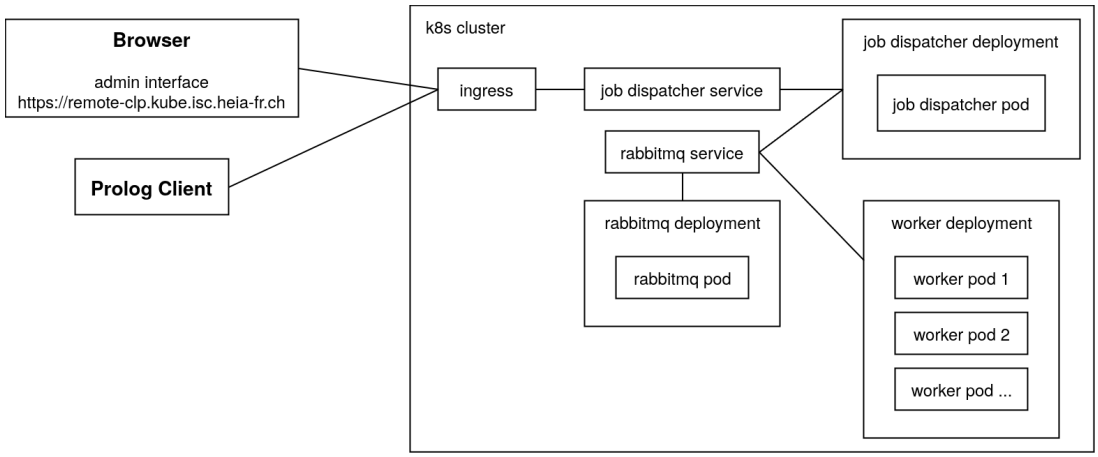

Figure 2: Kubernetes deployment architecture

3.2.2. Client-server communication

Figure 3 shows the communication protocol between the client and the server.

The client sends a request to the job dispatcher. The job dispatcher authenticates the client and sends the request to the message queue. The message queue distributes the request to the workers. The workers solve the request and send the result back to the job dispatcher. The job dispatcher sends the result back to the client.

During the process, the client can poll the results endpoint to get the status of the request. When the request is finished, the status endpoint will return the data of the result or an error message.

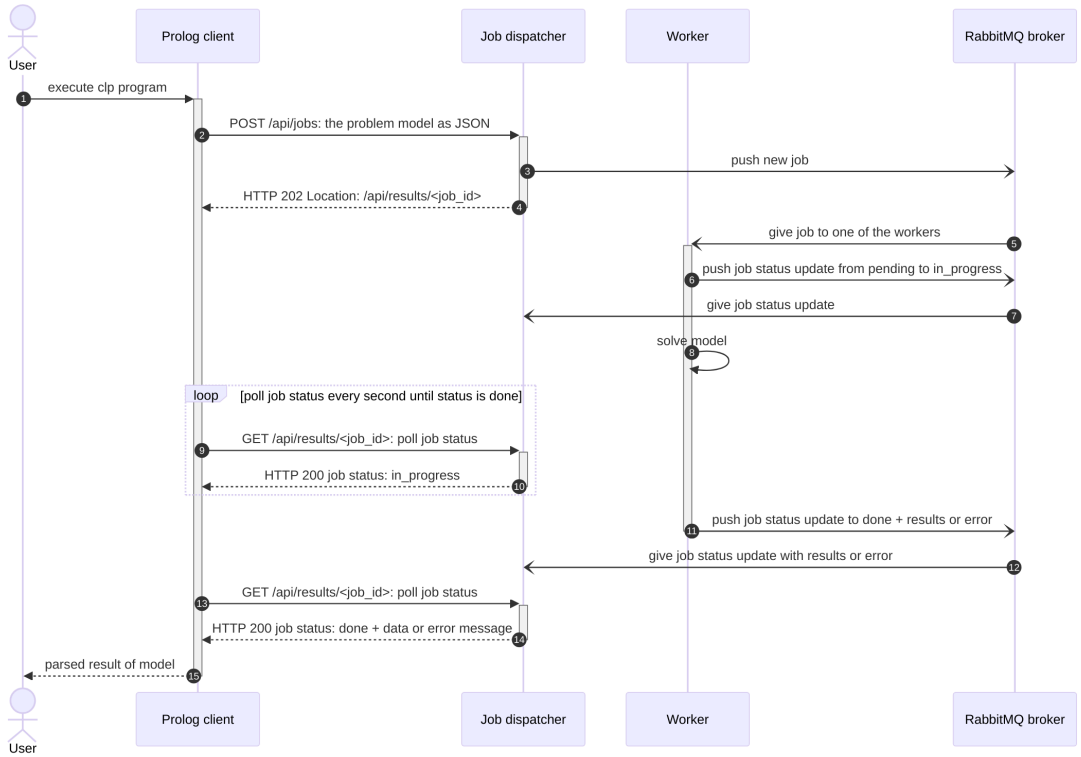

Figure 3: Client-server communication protocol

3.2.3. Request and response data structure

Listing 13 shows an example of the request data structure that the client sends to the server. The request is a JSON object that contains the options, the variables and the constraints of the problem.

The JSON object has the following structure:

- `options`: an array of options that the user can specify. The options are used to specify the objective of the problem (minimize or maximize) or limits on the solver (time and number of solutions).
- `id`: a unique identifier for the problem (UUIDv4).
- `variables`: an array of variables that the user can specify. Each variable has an upper bound, a lower bound and a unique identifier (the bounds are optional, if not specified the bounds are set to negative and positive infinity).
- `constraints`: an array of constraints that the user can specify. Each constraint has a unique identifier, a type and a value. The value is a list of variables or an arithmetic expression. There are two types of constraints: `all_different` and `arithmetic`.
  - `all_different`: specifies that all the variables must have different values. The value is a list of variables.
  - `arithmetic`: specifies an arithmetic constraint. The value is an arithmetic expression. An arithmetic expression is a tree with operators and literals. The operators are `+`, `-`, `*`, `//` and `mod`. The top level operator is the comparison operator (`>`, `<`, `==`, `>=`, `<=` and `!=`). The nodes of the tree are variables or integer literals.

```
{
    "options":  [  {"value":"dcba0bcd-f15b-4848-a4f3-ee8663292512",
"type":"max"} ],
  "id":"44a67e0f-f325-4718-87f8-fccd0ec73081",
  "variables": [
    {"ub":5, "lb":0, "id":"4d7ae8ee-c214-47b8-91bb-946b3546c245"},
    {"ub":5, "lb":0, "id":"dcba0bcd-f15b-4848-a4f3-ee8663292512"},
    {"ub":5, "lb":0, "id":"9e574287-536e-4073-9ef0-4f9981041e7c"}
  ],
  "constraints": [
    {
      "id":"d8395094-ec1d-4e61-8954-5406ed6bd8c0",
      "type":"all_different",
      "value": [
        "4d7ae8ee-c214-47b8-91bb-946b3546c245",
        "dcba0bcd-f15b-4848-a4f3-ee8663292512",
        "9e574287-536e-4073-9ef0-4f9981041e7c"
      ]
    },
    {
      "id":"3b027706-cfb8-47d0-8ec2-1344410a1601",
      "type":"arithmetic",
      "value": {
        "type":"operator",
        "value":"<=",
        "children": [
          {
            "type":"variable",
            "value":"4d7ae8ee-c214-47b8-91bb-946b3546c245"
          },
          {
            "type":"operator",
            "value":"*",
            "children": [
              {"type":"literal", "value":2},
              {
                "type":"variable",
                "value":"dcba0bcd-f15b-4848-a4f3-ee8663292512"
              }
            ]
          }
        ]
      }
    }
  ]
}
```

Listing 13: Example of a request data structure

The result data structure that the server sends back to the client is a simple JSON array of objects. Each object represents a solution to the problem. The entries of the

object are the variables UUIDs as keys and their values as values. In the case of an error, the server sends back a JSON object with the error message.

3.3. Testing strategy

The testing strategy for the client library is to use the example programs that were presented in the previous section. These programs will be used to test the client library and ensure that it works as expected. Since this is an end to end test of the client library, it is difficult to automate it completely. The user must manually run the programs and check the results.

The example programs will contain some test cases so that we can simply execute the test predicates to check if the client library works as expected. Naturally these programs will be used to test the remote CLP service as well.

The following test cases will be used to test the client library:

- N-queens problem:
  - Test different values of N and check if the solutions are correct.
  - Test the amount of solutions found for different values of N.
- Pythagorean triplets problem:
  - Test different values of N and check if the solutions are correct.
  - Test the amount of solutions found for different values of N.
- Optimization problem: verify that the solution is correct.
- Word puzzle: verify that the solution is correct.

In addition to these tests we will also perform some performance tests to check the scalability of the client library. This is done by measuring the time it takes to solve a problem for different values of N (where applicable). The times are compared to the times gotten from CLPFD library [10]. Since the example programs have the same syntax as the CLPFD library, it is easy to compare the times.

This testing strategy is quite simple but should be sufficient to ensure that the client library works as expected. In the future, more specific tests could be added on the individual components of the application.

Section 4
Implementation

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

4.1. Job dispatcher

The job dispatcher is the entry point of the service. It is responsible for authenticating the client distributing the requests to the workers. The job dispatcher is implemented in Go with the Fiber framework.

4.1.1. Internal Structure

The job dispatcher code is split into several files:

- `main.go`: the main file that starts the server and sets up the routes with their handlers.
- `types.go`: contains the data structures that are used in the job dispatcher such as a job request and a job response.
- `broker.go`: contains the code that interacts with the message broker.
- `config.go`: contains the code that reads the configuration from the environment variables.

The job dispatcher uses the Fiber framework to handle the HTTP requests. Fiber is a fast and lightweight web framework for Go that is easy to use and deploy. There is no need for a database in the job dispatcher since it only forwards the requests to the message queue.

The job dispatcher uses a thread-safe map to store the jobs that are being processed. When a job is received, the job dispatcher stores the job in the map. The job dispatcher then sends the job to the message queue and waits for the result. When the result is received, the job dispatcher updates the job in the map and sends the result back to the client. The unique identifier generated by the client is used to identify the job during the process.

Since the thread-safe map provided by Go doesn't support generics, the map is implemented as a map of interfaces. To simplify the code, the map type was wrapped into an new type `JobResultMap` that provides methods to get and set jobs.

The model of the job itself is not parsed in the job dispatcher. The job is sent to the message queue as is. The message queue is responsible for parsing the job and sending

it to the workers. The job dispatcher only stores the UUID of the job, its status and the result / error.

4.1.2. Authentication

The job dispatcher uses a JWT token [2] based authentication system. The user must provide a token in the request header to authenticate. The token is generated by the administration interface. The administration password is set in the environment variables.

The job dispatcher uses the `golang-jwt` library to handle the JWT tokens. The token is signed with a secret key that is set in the environment variables. The token contains the user's username (for future rate limiting / user metrics) and the expiration date. The expiration date is chosen by the administrator when the token is generated.

To ensure that the routes that require authentication are protected, the job dispatcher uses middleware (provided by the Fiber library) that checks the token in the request header. If the token is not valid or has expired, the middleware returns a 401 Unauthorized status code. The order in which the middleware is used is important. The middleware that checks the token must be used before the middleware that handles the routes. So routes that don't require authentication need to be defined before the middleware.

In addition to authentication, the results endpoint is protected by a check that verifies that the user is the owner of the job. This is done by checking the UUID of the job against the UUID in the token. If the UUIDs don't match, the job dispatcher returns a 403 Forbidden status code.

4.1.3. Message broker usage

The job dispatcher uses the Rabbit MQ message broker to distribute the requests to the workers. The job dispatcher sends the job to the message queue and waits for the result. The message queue is responsible for parsing the job and sending it to the workers. The workers solve the request and send the result back to the message queue. The message queue then sends the result back to the job dispatcher.

The message broker is has two queues: one for the jobs and one for the results / status updates. The job dispatcher sends the job to the jobs queue and waits for the result / status updates in the results queue. The workers consume the jobs queue and produce the results queue.

The `streadway/amqp` library is used to interact with the Rabbit MQ message broker. The library provides a simple and easy to use API to send and receive messages from the message queue. A wrapper is used to simplify the code and provide a simple API to send and receive messages. The `main.go` initializes the connection to the message queue and passes a callback that gets called when a message arrives in the job results queue. In the `broker.go` file, the queue is declared and a consumer is started to receive the

results. Using this connection, the endpoint that creates a job can then send the job to the message queue and wait for the result.

4.1.4. Admin interface

The admin interface is a simple web interface that allows the administrator to generate JWT tokens. The admin interface is a simple static HTML page that uses JavaScript to send requests to the job dispatcher.

Figure 4 shows the admin interface. The page contains a form with 3 fields: the username, the expiration date and the administration password. When the form is submitted, the JavaScript code sends a POST request to the job dispatcher with the data from the form. The job dispatcher checks the administration password and generates a JWT token with the username and the expiration date. The token is then sent back to the admin interface and displayed to the user.

For styling, the admin interface uses the bootstrap CSS framework. The JavaScript code uses the `fetch` API to send the requests to the job dispatcher.

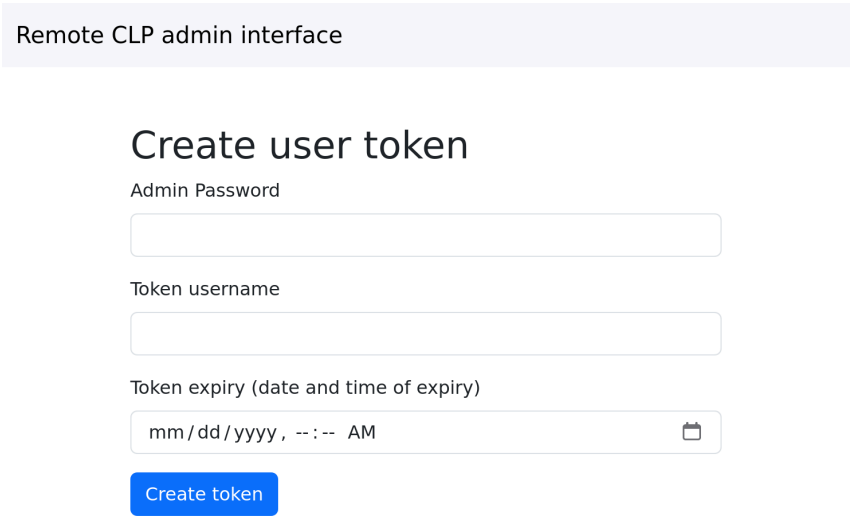

Figure 4: Admin interface

4.2. Worker

The worker is responsible for solving the requests that are sent by the job dispatcher. The worker is implemented in Python with the Google OR-Tools library.

4.2.1. Internal Structure

The worker code is split into several files:

- `app.py`: the main file that starts the worker and sets up the connection to the message queue.

- `model_parser.py`: contains the code that parses the job and creates the model for the solver.
- `broker.py`: contains the code that interacts with the message queue.

The worker retrieves the job from the message queue and parses it with the `model_parser.py` module the result of this is a Google OR-Tools model for the CP-SAT solver. The model is then solved and the result parsed back to a JSON object. This object is then sent back to the message queue.

4.2.2. Model parsing

For the model parsing, the worker uses the Google OR-Tools library. The library provides a simple and easy to use API to create and solve constraint programming problems. The worker uses the `model_parser.py` module to parse the job and create the model for the solver.

First the variables are created with the `new_int_var` method on the model. The variables are stored in a dictionary with their UUID as the key. The UUID is also used as name for the variable. If the bounds are not specified, the bounds are set to the negative and positive 32 bit integer limits (it is the recommended way to create a variable in the Google OR-Tools library).

The constraints are then parsed. If the constraint is an `all_different` constraint, the `add_all_different` method is called on the model with the list of variables. If the constraint is an `arithmetic` constraint, the tree is parsed recursively and the corresponding method is called on the model. The tree is parsed by a recursive function that combines the children of the tree with the operator of the parent node and produces a `LinearExpr` object (the case of non-linear constraints will be discussed in a later paragraph). The top level operator is the comparison operator (`>`, `<`, `==`, `>=`, `<=` and `!=`) which turns the `LinearExpr` object into a `BoundedLinearExpr` object. The nodes of the tree are variables or integer literals. The `BoundedLinearExpr` object is then used to create the constraint with the corresponding method on the model.

The Google OR-Tools library provides overloads for the arithmetic operators. If a variable is used in a linear arithmetic expression (i.e. variable times literal or variable + variable, see Listing 14), the library automatically creates a `LinearExpr` object for the variable. Since this only works for linear expressions, the worker manually creates the constraints using the `add_multiplication_equality`, `add_division_equality`, etc. methods for non-linear expressions (see Listing 14).

```python
model.Add(x + 2 * y <= 10) # ok
model.Add((x + 2) * y <= 10) # not ok

aux = model.NewIntVar(0, 10, 'aux')
model.AddMultiplicationEquality(aux, x + 2, y)
model.Add(aux + 2 <= 10) # ok
```

Listing 14: Example of arithmetic expressions in Google OR-Tools

4.3. Prolog client library

The Prolog client library is a library that allows the user to access the remote CLP service from Prolog. The client library is implemented in SWI-Prolog.

4.3.1. Internal Structure

The client library code is contained in a single file `remote_clp.pl`.

4.3.2. Variable tracking and parsing

As already mentioned in Section 2.4, 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. The client library uses the `get_remote_clp_attr` and `put_remote_clp_attr` to manage an attribute for the `remote_clp` module to the variable. The term used as an attribute for the variable is used as follows `var(Uuid, Lb, Ub)`.

The `var/3` term is then used to create the JSON object that is sent to the server. The variables that are included in the list given to the labeling predicates are parsed to JSON objects. If there are any other variables that are involved in a constraint that involves a variable from the labeling list, then these variable are also implicitly added to the model.

If an arithmetic constraint uses a variable that doesn't have the `var/3` attribute, the client library implicitly adds the variable to the model with infinite bounds.

4.3.3. Constraint tracking

To track the constraints, the client library uses a global list of constraints. The constraints are stored in a list of `constraint/2` terms. The `constraint/2` term contains the JSON object representing the constraint and a list of `var/3` terms representing the involved. The constraints that involve the variables included in the list given to the labeling predicates are parsed to JSON objects.

After the solution is received, the client library cleans up the global list of constraints. The constraints that are involved in the solution are removed from the list. This is done to prevent the list from growing indefinitely. This enables us to call the labeling

predicates multiple times for different variables and constraints without the need to restart the client library. This however is not a perfect solution since the list can still grow indefinitely if the user doesn't call the labeling predicates.

4.3.4. HTTP requests

The client library uses the `http_post` and `http_get` predicates from the `http/http_client` library to send the requests to the server. The requests are sent as JSON objects. The client library uses the `http/http_json` library to parse and serialize the JSON objects.

The client library adds the token to the request header. The token (and the URL of the service) is stored in a SWI-Prolog setting (special global variable) that is set with the `api_config/1` predicate. If the request succeeds and the status code is 202, the client library retrieves the location header from the response and polls the status endpoint until the job status is done.

As already mentioned in Section 2.3 the client library uses the `thread_wait` predicate to poll the status endpoint. The client library uses the `http_get` predicate to send the request to the status endpoint and parse the JSON object that is received.

The JSON object that is received from the server is parsed and the variables are unified with the values. In the case where there are multiple solutions, the client library backtracks to find all the solutions.

4.3.5. Tests cases on example programs

TODO

4.4. Kubernetes deployment

The remote CLP service is deployed on a Kubernetes cluster. The service is split into three components: the job dispatcher, the message queue and the workers. Each component is deployed as a separate pod in the Kubernetes cluster.

4.4.1. Gitlab CI/CD

The deployment of the service is automated with Gitlab CI/CD. The CI/CD pipeline is defined in the `.gitlab-ci.yml` file. The pipeline consists of several stages: build and deploy. In the build stage, the Docker images are built and pushed to the Gitlab container registry of the project. In the deploy stage, the Kubernetes manifests are applied to the Kubernetes cluster.

```yaml
stages:
  - build
  - deploy

build-images:
  stage: build
  image: docker:24.0.3
  services:
    - docker:24.0.3-dind
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
          -   docker   build   -t   $CI_REGISTRY_IMAGE/job_dispatcher:latest
-f     remote_clp_service/job_dispatcher/Dockerfile     remote_clp_service/
job_dispatcher
    - docker build -t $CI_REGISTRY_IMAGE/worker:latest -f remote_clp_service/
worker/Dockerfile remote_clp_service/worker
    - docker push $CI_REGISTRY_IMAGE/job_dispatcher:latest
    - docker push $CI_REGISTRY_IMAGE/worker:latest

k8s-deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: [ "" ]
  stage: deploy
  needs:
    - build-images
  script:
    - echo "Deploying application to K8s..."
    - kubectl config set-cluster local --server="https://rancher.kube.isc.
heia-fr.ch/k8s/clusters/local"
    - kubectl config set-credentials local --token="$KUBE_TOKEN"
    - kubectl config set-context local --user=local --cluster=local
    - kubectl config use-context local
    - kubectl -n remote-clp apply -f ./kubeconfig
    - kubectl -n remote-clp rollout restart deploy remote-clp-job-dispatcher
remote-clp-worker
    - echo "Application successfully deployed."
```

Listing 15: Example of arithmetic expressions in Google OR-Tools

4.4.2. Docker compose for local testing

To test the service locally, a `docker-compose.yml` file is provided. The file defines the services that are needed to run the service locally. The services are the job dispatcher, the message queue and the workers. The services are defined with the Docker images built by the Dockerfiles in the respective folder (the same ones build in the CI/CD pipeline).

To run the service locally, the user must create a `.env` file with the environment variables that are needed by the services. The `.env` file must contain the administrator password

and the secret key for the JWT tokens. The user can then run the `docker-compose up` command to start the services.

4.4.3. Deployment of the job dispatcher

The job dispatcher is deployed as a Kubernetes deployment with a single replica. The service is exposed to the internet through the ingress. The job dispatcher uses a service to expose the deployment to the message queue and the workers.

The docker image of the job dispatcher is built with the Dockerfile in the `remote_clp_service/job_dispatcher` folder. The image is pushed to the Gitlab container registry of the project. The Kubernetes deployment uses the image from the container registry. The image is built in 2 steps: first the dependencies are installed and compiled and then the final image is built with the binary.

The job dispatcher uses environment variables to configure the connection to the message queue, the administration password, the JWT secret and the port for the server. The environment variables are set in the Kubernetes manifest. The administrator password and the JWT secret are stored in the Kubernetes secrets.

4.4.4. Deployment of the workers

The workers are deployed as a Kubernetes deployment with multiple replicas. The workers use a service to expose the deployment to the message queue. The workers use the image built by the Dockerfile in the `remote_clp_service/worker` folder. The image is pushed to the Gitlab container registry of the project. The image is simple and only contains the Python code and the dependencies.

The workers use environment variables to configure the connection to the message queue. The environment variables are set in the Kubernetes manifest.

4.4.5. Deployment of the message broker

The message broker is deployed as a Kubernetes deployment with a single replica. The message broker uses a service to expose the deployment to the job dispatcher and the workers. The message broker uses the Rabbit MQ image from the Docker Hub.

Persistent volumes are used to store the data of the message broker. The persistent volumes are mounted to the `/var/lib/rabbitmq` and `/var/log/rabbitmq` directories of the container. The persistent volumes are defined in the Kubernetes manifest. The docker compose file for local testing also uses persistent volumes.

Section 5

Results

Section 6
Conclusion

In this project, we have implemented a Prolog client library and a remote CLP service that allows the user to solve constraint programming problems with the Google OR-Tools library. The client library is easy to use and feels similar to other CLP libraries that are available in Prolog. The client library supports finite domain variables, arithmetic constraints (liner and non-linear ones) and all different constraints. The client library is tested with example programs that show how to use the library to solve different problems.

6.1. Challenges

The main challenges of the project were:

6.1.1. Routes require authentication where they shouldn't

The routes that require authentication are protected by middleware that checks the token in the request header. The middleware is used to protect the routes that require authentication. The middleware checks the token in the request header and returns a 401 Unauthorized status code if the token is not valid or has expired. The order in which the middleware is used is important. The middleware that checks the token must be used before the middleware that handles the routes. So routes that don't require authentication need to be defined before the middleware.

During the implementation of the job dispatcher, the middleware for authentication was placed after the middleware that handles the routes. This caused the routes that don't require authentication to be protected by the middleware. This was fixed by moving the middleware for authentication before the middleware that handles the routes.

6.1.2. Global constraint list is filling up

The client library uses a global list of constraints to keep track of the constraints that are involved in the problem. The constraints are stored in a list of `constraint/2` terms. The `constraint/2` term contains the JSON object representing the constraint and a list of `var/3` terms representing the involved variables. The constraints that involve the variables included in the list given to the labeling predicates should be parsed to JSON objects.

In the early implementation of the client library, the global list of constraints was not filtered before sending the request to the server. This caused the list to grow indefinitely

and the server to return an error. This was fixed by filtering the list of constraints before sending the request to the server. Additionally, the client library cleans up the global list of constraint used in the labeling predicates after the solution is received.

This reduces the risk of the list growing indefinitely but doesn't prevent it. The list can still grow indefinitely if the user doesn't call the labeling predicates.

6.1.3. RabbitMQ channel closes during long running requests

The RabbitMQ channel closes during long running requests. The connection type (`BlockingConnection`) that is used in the worker doesn't automatically reconnect to the message queue during the execution of the solver. This causes the worker to lose the connection to the message queue and when the solver is finished, the worker can't send the result back to the message queue.

To fix this issue, the worker starts a thread that manually sends a heartbeat to the message queue every 3 minutes. The heartbeat is a simple message that is sent to the message queue. The message is ignored by the message queue but keeps the connection alive. This prevents the channel from closing during the execution of the solver.

6.1.4. Duplicate solutions in the results

The worker sends the results back to the message queue as soon as the solver finds a solution. The worker doesn't check if the solution is a duplicate of a previous solution. This causes the results to contain duplicate solutions. The solver uses a solution callback class that gets called every time a solution is found. In the case of the worker solution callback class stores the solutions in a list.

To fix this issue, the worker iterates over the list of solutions after the solver is finished and removes the duplicates.

6.2. Future work

The project could be extended in several ways:

6.2.1. Add more constraints and arithmetic operators to the client library

Right now the client library only supports finite domain variables, arithmetic constraints and all different constraints. The client library could be extended to support more constraints and arithmetic operators. The client library could also be extended to support more complex constraints using reification.

The CP-SAT solver of the Google OR-Tools library supports a wide range of constraints and arithmetic operators that are not currently implemented. The client library could be extended to support these constraints and operators.

A possible extension would be to add support for another solver of the Google OR-Tools library, like the CP solver, the MIP solver or even use another library like the Gecode. However such a changes would require a complete rework of the worker because it was not designed for it.

6.2.2. Improve testing strategy

The testing strategy of the client library could be improved. Currently the testing strategy is manual and relies on the user to run the example programs and check the results. The testing strategy could be improved by adding more automated tests on the individual components of the application. For example unit tests on the job dispatcher, the worker and the client library directly.

Individual unit tests are easier to automate and provide more coverage of the code. The unit tests could be run automatically with a continuous integration system like Gitlab CI/CD.

6.2.3. Add rate limiting and user statistics to the job dispatcher

The job dispatcher could be extended to support rate limiting and user statistics. The job dispatcher could keep track of the number of requests that are sent by each user and limit the number of requests that a user can send in a given time period. The job dispatcher could also keep track of the number of requests that are solved by each user and provide statistics on the number of requests that are solved by each user.

Right now the administration interface only allows the administrator to generate JWT tokens. The administration interface could be extended to allow the administrator to revoke tokens and view statistics on the number of requests that are sent and solved by each user. This could allow for detecting abusive usage (and revoke the tokens of these users) of the service and provide insights on the usage of the service.

6.2.4. Add support for GNU Prolog in the client library

The client library is currently implemented in SWI-Prolog. The client library could be extended to support GNU Prolog. The client library could be extended to support both SWI-Prolog and GNU Prolog. The client library could be extended to support more Prolog implementations in the future.

GNU Prolog is another popular Prolog implementation that is widely used in the Prolog community, however it doesn't have the same amount of features like SWI-Prolog. GNU Prolog doesn't support HTTP requests out of the box, it would need to be implemented by using system sockets to make TCP connections. But other than that it should be doable.

6.3. Personal opinion

The project was a great learning experience. I learned a lot about constraint programming and the Google OR-Tools library. I also learned a lot about Go and the Fiber framework. The project was challenging but rewarding. But most of all it was the first time using Prolog in a project and I was positively surprised by the amount of quality of life features it has. I'm happy with the result and I think the project turned out well.

Declaration of honor

In this project, we used generative AI tools, namely Github Copilot [5] for coding and Claude AI [1] 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

[1] Anthropic. 2024. An LLM based AI chat bot. Retrieved from https://claude.ai/

[2] auth0. 2024. A JSON based token standard for representing security claims. Retrieved from https://jwt.io/

[3] Broadcom. 2024. A popular message broker often used for asynchronous message passing. Retrieved from https://www.rabbitmq.com/

[4] Daniel Diaz. 2009. Word puzzle CLP program provided as an example for the GNU Prolog CLP library. Retrieved from https://github.com/maandree/gprolog/blob/master/examples/ExamplesFD/alpha.pl

[5] Github. 2024. An AI based developer tool for code generation. Retrieved from https://github.com/features/copilot

[6] Google. 2023. Google OR-Tools MathOpt. Retrieved from https://developers.google.com/optimization/math_opt

[7] Google. 2024. A lightweight and fast systems programming language. Retrieved from https://go.dev/

[8] Jordan Vésy. 2022. *Alternative CLP Engines.*

[9] József Sallai. 2024. An Express-inspired web framework written in Go. Retrieved from https://gofiber.io/

[10] Markus Triska. 2024. SWI-Prolog clpfd library. Retrieved from https://www.swi-prolog.org/man/clpfd.html

[11] Noah Godel. 2024. *Prolog remote constraint logic programming - Specification.*

[12] Python Software Foundation. 2024. A simple and popular interpreted programming language. Retrieved from https://www.python.org/

Table of figures