Background

This thesis illustrates an approach of implementing a language server for the Nickel language which communicates with its clients, i.e. editors, over the open Language Server Protocol (in the following abbreviated as *LSP*). The current chapter provides the backgrounds on the technological details of the project. As the work presented aims to be transferable to other languages using the same methods, this chapter will provide the means to distinguish the nickel specific implementation details.

The primary technology built upon in this thesis is the language server protocol. The first part of this chapter introduces the LSP, its rationale and improvements over classical approaches, technical capabilities and protocol details. The second part is dedicated to Nickel, elaborating on the context and use-cases of the language followed by an inspection of the technical features Nickel is based on.

Language Server Protocol

Language servers are today's standard of integrating support for programming languages into code editors. Initially developed by Microsoft for the use with their polyglot editor Visual Studio Code[ˆhttps://code.visualstudio.com/] before being released to the public in 2016 by Microsoft, RedHat and Codeenvy, the LSP decouples language analysis and provision of IDE-like features from the environment used to write. Developed under open source license on GitHub[1], it allows developers of editors and languages to work independently on the support for new languages. If supported by both server and client, the LSP now supports more than 24 language features[ˆhttps://microsoft.github.io/language-server-protocol/specifications/specification-current/] including code completion, hover information, resolution of type and variable definitions, controlling document highlighting, formatting and more.

Motivation

Since its release, the LSP has grown to be supported by a multitude of languages and editors[@langservers [@lsp-website]], solving a long-standing problem with traditional IDEs.

Before the inception of language servers, it was the development platforms' individual responsibility to implement specialized features for any language of interest. This effectively causes a $M \times N$ where M editors have to implement support for N languages. Under the constraint of limited resources editors had to position themselves on a spectrum between specializing on integrated support for a certain subset of languages and being generic over the language providing only limited support. As the former approach offers a greater business value, especially for proprietary products most professional IDEs gravitate towards excellent (and exclusive) support for single major languages, i.e. XCode and Visual Studio for

[1] https://github.com/microsoft/language-server-protocol/

the native languages for Apple and Microsoft Products respectively as well as JetBrains' IntelliJ platform and RedHat's Eclipse. Problematically, this results in less choice for developers and possible lock-in into products less favorable but unique in their features for a certain language. The latter approach was taken by most text editors which in turn offered only limited support for any language.

Looking at popularity statistics[2] shows that except Vim and Sublime Text, both exceptional general text editors, the top 10 most popular IDEs were indeed specialized products. The fact that some IDEs are offering support for more languages through (third-party) extensions, due to the missing standards and incompatible implementing languages/APIs, does not suffice to solve the initial problem that developing any sort of language support requires redundant resources.

This is especially difficult for emerging languages, with possibly limited development resources to be put towards the development of language tooling. Consequently, community efforts of languages any size vary in scope, feature completeness and availability.

The Language Server Protocol aims to solve this issue by specifying a JSON-RPC[^Remote Procedure Call] API that editors (clients) can use to communicate with language servers. Language servers are programs that implement a set of IDE features for one language and exposing access to these features through the LSP, allowing to focus development resources to a single project, hence reducing the required work to bring language features of N languages from $M \times N$ to N.

JSON-RPC

JSON-RPC (v2) [@json-rpc] is a JOSN based lightweight transport independent remote procedure call protocol used by the LSP to communicate between language server and client.

The protocol specifies the general format of messages exchanges as well as different kinds of messages. The following snippet [#lst:json-rpc-req] shows the schema for request messages.

Listing 1 JSON-RPC Request

```
// Requests
{
  "jsonrpc": "2.0"
, "method": String
, "params": List | Object
, "id": Number | String | Null
}
```

`"jsonrpc" : "2.0"` A fixed value format indicator

[2] https://web.archive.org/web/20160625140610/https://pypl.github.io/IDE.html

`"method"` The name of the procedure called on the server
: May not start with `rpc.` which is an indicator for internal messages

`"params"` An optional set of parameters passed to the executed method.
: Parameters can be passed as a list of arguments or as a named dictionary.

`"id"` A (unique) identifier for the current message
: Used to answer client requests
: Messages without an `id` are considered to be *Notifications*

The main distinction in JSON-RPC are *Requests* and *Notifications*. Messages with an `id` field present are considered *requests*. Servers have to respond to requests with a message referencing the same `id` as well as a result, i.e. data or error. If the client does not require a response, it can omit the `id` field sending a *notification*, which servers cannot respond to, with the effect that clients cannot know the effect nor the reception of the message.

Responses as shown in [#lst:json-rpc-res], have to be sent by servers answering to any request. The `id` field has to match the one corresponding request message. If the called procedure was successful, its return value is encoded under the `return` key, while errors occuring during the call are recorded under the `error` key. Errors are represented as objects specifying the error kind using an error `code` and providing a human-readable descriptive `message` as well as optionally any procedure defined `data`.

Listing 2 JSON-RPC Response and Error

```
// Responses
{
  "jsonrpc": "2.0"
  "result": any
  "error": Error
, "id": Number | String | Null
}

// Error
{
    "code": Integer,
    "message": String,
    "data": any
}
```

Clients can choose to batch requests and send a list of request or notification objects. The server should respond with a list of results matching each request, yet is free to process requests concurrently.

JSON-RPC only specifies a message protocol, hence the transport method can be freely chosen by the application.

Commands and Notifications

File Notification

Diagnostics

Hover

Completion

Go-To-*

Symbols

code lenses

Shortcomings

Infrastructure as Code

Software defined Networks

Data oriented languages

Nickel

Gradual typing

Row types

Contracts