

Title

Federico Cristiano Bruzzone

Id. Number: 27427A

MSc in Computer Science

Advisor: Prof. Walter Cazzola

Co-Advisor: Dr. Luca Favalli



UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

Academic Year 2023-2024

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Background | 3 |
| 2.1 | Language Server Protocol | 3 |
| 2.1.1 | JSON-RPC | 4 |
| 2.1.2 | Command Specifications | 5 |
| 2.1.3 | Key Methods Overview | 7 |
| 2.2 | Language Workbenches | 11 |
| 2.3 | Type Systems | 11 |
| 2.4 | Software and Language Product Lines | 11 |
| 3 | Concept | 13 |
| 4 | Implementation | 15 |
| 5 | Evaluation | 17 |
| 6 | Related Work | 19 |
| 7 | Conclusions | 21 |

1

Introduction

2

Background

In this chapter, we provide an overview of the concepts and technologies that are relevant to the work presented in this thesis. We start by introducing the concept of language servers and the Language Server Protocol (LSP) in Section 2.1. We then discuss language workbenches in Section 2.2, type systems in Section 2.3, and software and language product lines in Section 2.4.

The goal of this chapter is to provide the reader with the necessary background knowledge to understand the work presented in the following chapters. We assume that the reader has a basic understanding of programming languages and software development.

2.1 Language Server Protocol

The Language Server Protocol¹ (LSP) is a protocol that allows for the communication between a language server and an editor or an IDE. The LSP is used to decouple the a language-agnostic editor or integrated development environment (IDE) from the language-specific features of a language server (see Listing 2.1). This allows for the development of language servers that can be used with multiple editors or IDEs. The LSP is based on the stateless JSON-RPC protocol and defines a set of messages that are used to communicate between the language server and the editor or IDE.

Usually, the LSP Clients are developed as plugins for popular editors or IDEs decreasing the effort to support a new language in a given editor. The LSP Clients are responsible for sending requests to the language server and processing the responses. The language server is responsible for providing language-specific features. The LSP defines a set of messages that are used to communicate between the language server and the editor or IDE. These messages include requests for code completion, code navigation, and code analysis, as well as notifications for changes to the document, diagnostics, and progress reports.

Language servers are *de facto* standard for providing language-specific features in editors and IDEs. The LSP is supported by a wide range of editors and IDEs, including Neovim², Visual Studio Code³, Eclipse⁴, and IntelliJ IDEA⁵. There are several language

¹<https://microsoft.github.io/language-server-protocol>

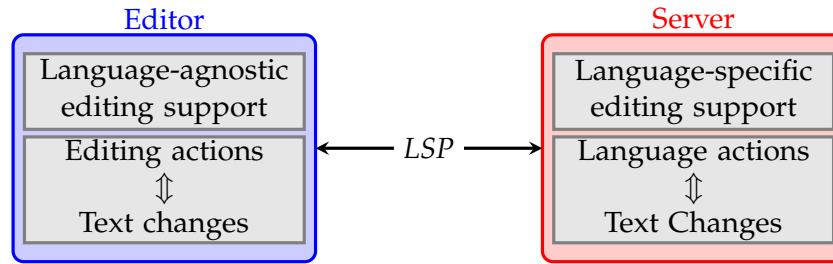
²<https://neovim.io/doc/user/lsp.html>

³<https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>

⁴https://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php

⁵<https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html>

2 Background



Listing 2.1. LSP approach to language support. Borrowed from [2].

servers available for popular programming languages, including Rust, TypeScript, Python, and Java and most of them are open-source⁶.

The LSP is initiated by Microsoft and is now an open standard that is maintained by the Language Server Protocol Working Group. It was designed for the use with the Visual Studio Code editor, but it has since been adopted by other editors and IDEs. The LSP is under open-source license and is available on GitHub⁷.

2.1.1 JSON-RPC

The LSP uses JSON-RPC to communicate between a language server and an editor. JSON-RPC (v2)⁸ is a stateless, light-weight remote procedure call (RPC) [1] protocol that uses JSON as the data format.

RPC is a protocol that allows a client to call a procedure on a remote server. The client sends a request to the server, and the server sends a response back to the client. The JSON-RPC protocol defines a set of messages that are used to communicate between the client and the server. These messages include requests, responses, and notifications. The JSON-RPC protocol is designed to be simple and easy to implement, making it well-suited for use in web applications and other distributed systems.

JSON-RPC is a JSON based implementation of the RPC protocol. It defines a set of rules for encoding and decoding JSON data, as well as a set of rules for *Request*, *Notification*, and *Response* messages. The messages are sent over a transport layer, such as HTTP or WebSockets. The JSON-RPC protocol is designed to be simple and easy to implement, making it well-suited for use in web applications and other distributed systems.

All messages refer to a *method* that is a string containing the name of the method to be called. The *params* field is an array or object containing the parameters to be passed to the method. Typically, messages are synchronous, meaning that the client waits for a response from the server before continuing. The *id* field is a unique identifier for the message, which is used to match requests with responses. However, the JSON-RPC protocol also supports asynchronous messages, known as notifications, which do not

⁶<https://microsoft.github.io/language-server-protocol/implementors/servers>

⁷<https://github.com/microsoft/language-server-protocol>

⁸<https://www.jsonrpc.org/specification>

require a response from the server. This is implemented by setting the *id* field to *null*, in which case the server does not send a response back to the client.

The JSON-RPC specification includes the ability for clients to batch multiple requests or notifications by sending them as a list. The server is expected to respond with a corresponding list of results for each request. Additionally, the server has the flexibility to process these requests concurrently.

2.1.2 Command Specifications

The LSP is defined on top of the JSON-RPC protocol described in section 2.1.1. In abstract terms, the LSP defines a set of command that can be sent between a client and a server. In the Language Server Protocol Specification⁹, these commands are divided into four categories: *Language Features*, *Text Document Synchronization*, *Workspace Features*, and *Window Features*.

Language Features

The *Language Features* provide the smarts of the language server. Usually, the client sends a request to the server to get information about the document as a tuple of *TextDocument* and *Position*. *Code comprehension* and *Coding Features* are the two main categories of commands in this category.

Here a brief description of the most important commands in this category:

- *textDocument/completion*: The completion request is sent from the client to the server to compute completion items at a given cursor position. Completion items are presented in the editor's user interface. If computing full completion items is expensive, servers can additionally provide a handler for the completion item resolve request (*completionItem/resolve*). This request is sent when a completion item is selected in the user interface.
- *textDocument/hover*: The hover request is sent from the client to the server to request hover information at a given text document position. Hover information typically includes the symbol's signature and documentation.
- *textDocument/definition*: The definition request is sent from the client to the server to resolve the definition location of a symbol at a given text document position.
- *textDocument/references*: The references request is sent from the client to the server to resolve project-wide references for the symbol denoted by the given text document position.
- *textDocument/documentHighlight*: The document highlight request is sent from the client to the server to resolve a document highlights for a given text document position. For programming languages, this usually highlights all references to the symbol denoted by the given text document position.

⁹<https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification>

Text Document Synchronization

The *Text Document Synchronization* commands are used to notify the server of changes to the document. Client support for `textDocument/didOpen`, `textDocument/didChange`, `textDocument/didClose` is mandatory. This includes the ability to fully and incrementally synchronize changes to the document, such as inserting, deleting, and replacing text.

Here a brief description of the most important commands in this category:

- `textDocument/didOpen`: The document open notification signals that the client is now managing a text document. The server should not read the document using its URI. Open notifications are balanced with close notifications, with only one open notification allowed at a time for a document. The server's ability to fulfill requests is unaffected by a document's open or closed status.
- `textDocument/didChange`: The document change notification is sent from the client to the server to signal changes to a text document. In response, the server should compute a new version of the document's content. The server should not rely on the client to send a specific sequence of change events. The server is free to compute the new version of the document on the fly.
- `textDocument/didClose`: The document close notification is sent from the client to the server when the document is no longer managed by the client. The document's URI is no longer valid and the server should not resolve the document using the URI.
- `textDocument/didSave`: The document save notification is sent from the client to the server when the document is saved. The notification is sent after the document has been saved.

Workspace Features

The *Workspace Features* category includes commands that allow the client to interact with the workspace. The workspace is the collection of open documents and the client's configuration. The workspace commands are used to manage the workspace, such as symbol search, workspace configuration and client configuration.

Here a brief description of the most important commands in this category:

- `workspace/symbol`: The workspace symbol request is sent from the client to the server to list project-wide symbols matching the query string. The request can be used to populate a list of symbols matching the query string in the user interface.
- `workspace/configuration`: The workspace configuration request is sent from the client to the server to fetch configuration settings from the server. The request can fetch configuration settings from the client's workspace or from the server's configuration.
- `workspace/didChangeConfiguration`: The configuration change notification is sent from the client to the server to signal changes to the client's configuration settings. The server should use the new configuration settings to update its

behavior.

- `workspace/didChangeWorkspaceFolders`: The workspace folder change notification is sent from the client to the server to signal changes to the workspace's folders. The server should use the new workspace folders to update its behavior.

Window Features

The Window Features category includes commands that allow the client to interact with the window. The window is the client's user interface, such as the editor, the sidebar, and the status bar. The window commands are used to manage the window, such as showing messages, showing notifications, and showing progress.

Here a brief description of the most important commands in this category:

- `window/showMessage`: The show message notification is sent from the server to the client to show a message to the user. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.
- `window/showMessageRequest`: The show message request is sent from the server to the client to show a message to the user and get a response. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.
- `window/logMessage`: The log message notification is sent from the server to the client to log a message. The message can be logged in a variety of ways, such as a console, a file, or a database.
- `window/showMessageRequest`: The show message request is sent from the server to the client to show a message to the user and get a response. The message can be shown in a variety of ways, such as a dialog box, a status bar, or a notification.

2.1.3 Key Methods Overview

Six *key methods* have been identified by langserver organization¹⁰ as the most important methods to be implemented by a language server. These capabilities are:

1. **Diagnostic Analyze** source code — Parse and Type-check the source code to provide diagnostics.
2. **Workspace/Document Symbols** — List all symbols in the workspace.
3. **Jump to Definition** — Find and jump to the definition of a symbol.
4. **Find References** — List all usages of a symbol.
5. **Hover Information** — Show information about a symbol at the cursor.
6. **Code Completion** — Provide completions for a symbol at the cursor.

Diagnostic Analysis

The diagnostic analysis is the process of parsing and type-checking the source code to provide diagnostics. The diagnostics are used to identify errors and warnings in the

¹⁰<https://langserver.org>

2 Background

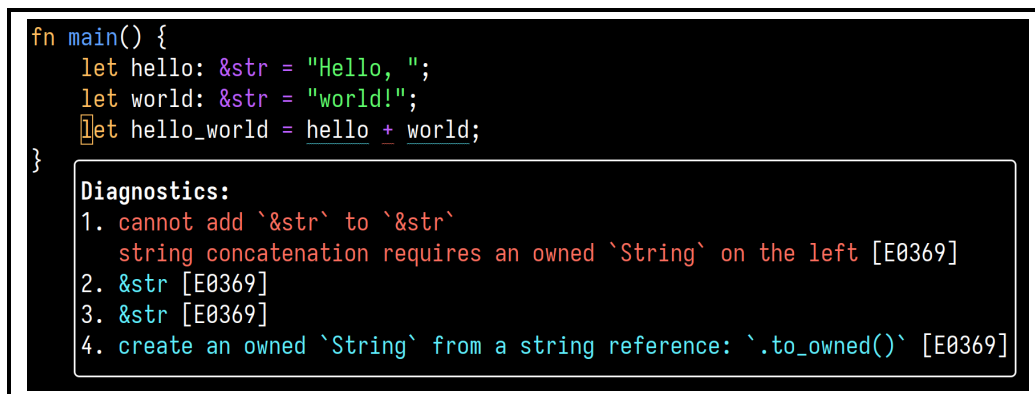


Figure 2.1. Showing diagnostics in Neovim generated by the Rust Language Server..

source code, such as syntax errors, type errors, and unused variables. The diagnostics are presented to the user in the editor's user interface, such as a list of errors and warnings in the sidebar.

File update and diagnostic analysis are passive. They are sent as notifications in order to avoid blocking communication between the client and the server.

In the figure 2.1 we can see an example of diagnostics in Neovim generated by the Rust Language Server. It shows a list of errors and suggestions caused by the concatenation of two `&str` variables.

Workspace/Document Symbols

RPC Method: `textDocument/workspaceSymbol` or `textDocument/documentSymbol`

This feature is defined as both a *Language Feature* and *Workspace Feature*.

The `textDocument/workspaceSymbol` request is sent from the client to the server to list project-wide symbols matching the query string. The request can be used to populate a list of symbols matching the query string in the user interface. The granularity of the listed symbols depends on the language server implementation.

The difference between `textDocument/workspaceSymbol` and `textDocument/documentSymbol` is that the first one takes into account the visibility of the symbols in the workspace showing only the public ones. The second one shows all symbols in the document.

Jump to Definition

RPC Method: `textDocument/definition`

The code navigation is an important feature of the LSP.

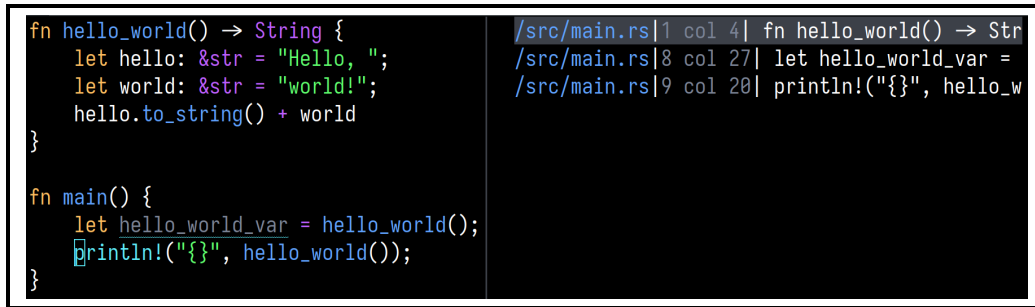


Figure 2.2. Finding references in Neovim generated by the Rust Language Server..

The textDocument/definition request is sent from the client to the server to resolve the definition location of a symbol at a given text document position. The server should return the location of the symbol's definition, such as the file path and line number.

In the figure 2.3 we can see that at the top of floating window there is the signature of the function `hello_world` meaning that the goind to definition will take us to the function definition.

Find References

RPC Method: `textDocument/references`

The retrieval of references is the inverse of the jump to definition explained in the previous section (2.1.3).

The textDocument/references request is sent from the client to the server to resolve project-wide references for the symbol denoted by the given text document position. The server should return a list of references to the symbol, such as the file path, line number, and the column number.

In the figure 2.2 we can see that the function `hello_world` is being referenced twice in the main function and another occurrence is the function definition. In fact, three references are shown in the right window.

Hover Information

RPC Method: `textDocument/hover`

The hover information is a feature that shows information about a symbol at the cursor. The information typically includes the symbol's signature and documentation. This is triggered by the user hovering the mouse over a symbol in the editor or by pressing a keybinding.

A request is sent from the client to the server to request hover information at a given text document position. The server should return hover information for the symbol at

2 Background



Figure 2.3. *Hover information in Neovim generated by the Rust Language Server.*

the given position, such as the symbol's signature and documentation.

In the figure 2.3 we can see that the function `hello_world` is being hovered and the signature of the function is shown in the floating window. The signature is `fn hello_world() -> String` and the documentation, written in the comment above the function, is also shown.

Code Completion

RPC Method: `textDocument/completion`

The code completion is a feature that provides completions for a symbol at the cursor. The completions are presented in the editor's user interface. If computing full completion items is expensive, servers can additionally provide a handler for the completion item resolve request (`completionItem/resolve`). This request is sent when a completion item is selected in the user interface.

Usually, the completion is triggered by the user typing a character or pressing a keybinding in proximity to a character, such as `.`, `::`, or `->`.

In the figure 2.4 we can see that the code completion is triggered by the user typing `to_` and pressing `<C-n>` in the editor. Two float windows are shown with the completion items. The first one shows the options for the `to_` function associated with the `&str` type and the second one shows the documentation of the selected completion item.

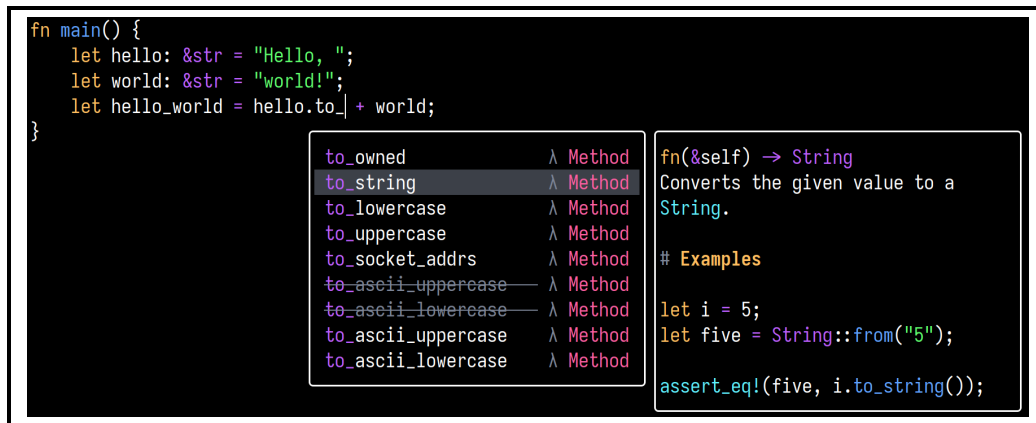


Figure 2.4. Code completion in Neovim generated by the Rust Language Server..

2.2 Language Workbenches

2.3 Type Systems

2.4 Software and Language Product Lines

3

Concept

4

Implementation

5

Evaluation

6

Related Work

7

Conclusions

Bibliography

- [1] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [2] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18*, page 370–380, New York, NY, USA, 2018. Association for Computing Machinery.