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

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

Contents

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.

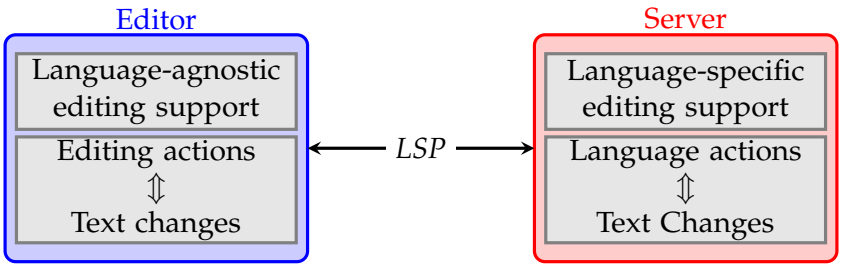
2.1 Language Server Protocol

The Language Server Protocol[1] (LSP) is a protocol that allows for the communication between a language server and an editor or an IDE. The LSP is used to decuple the a language-agnotic 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 decresing 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 Noevim[2], Visual Studio Code[3], Eclipse[4], and IntelliJ IDEA[5]. 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
[4] https://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php
[5] https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html



Listing 2.1. *LSP approach to language support. Borrowed from [6].*

servers available for popular programming languages, including Rust, TypeScript, Python, and Java and most of them are open-source[6].

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 GitHub[7].

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) [2] 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 Specification[9], 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 organization[10] 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

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. Thay 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 coused 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.

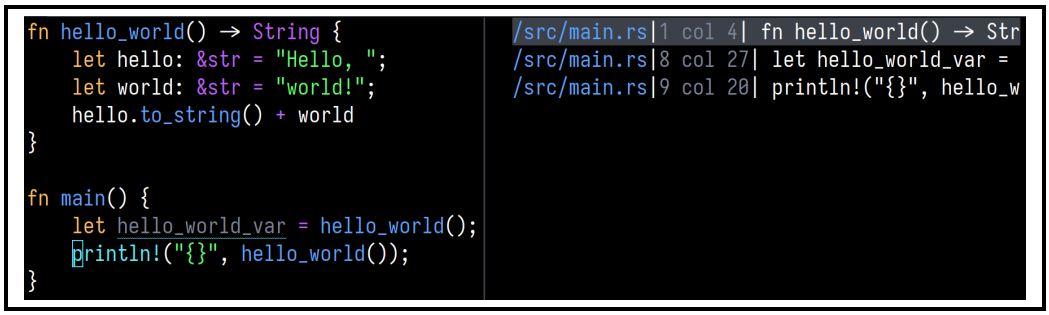


Figure 2.2. *Finding references in Neovim generated by the Rust Language Server..*

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 occurence 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

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.

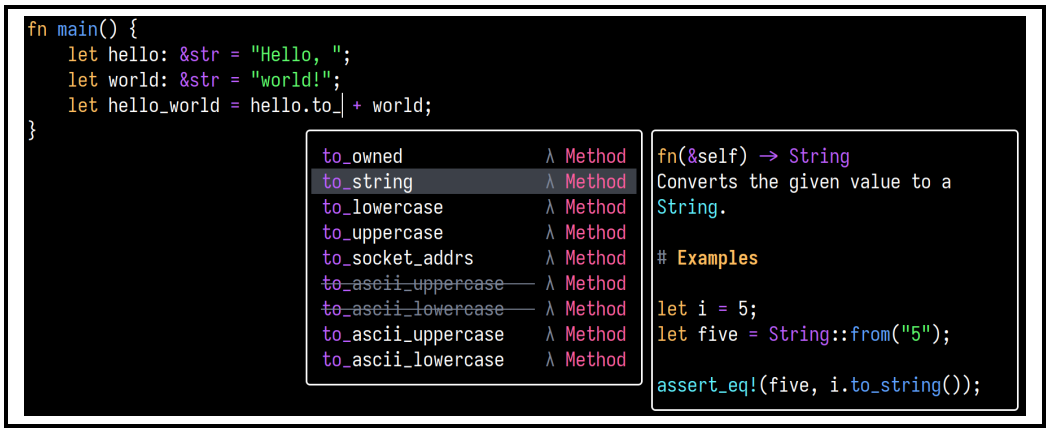


Figure 2.4. *Code completion in Neovim generated by the Rust Language Server..*

2.1.4 Approaches to Source Code Analysis in Language Servers

Language servers handle source code analysis using various methods, influenced significantly by the complexity of the language. The choice of approach affects how servers process file indexes, manage changes, and respond to requests.

The Language Server Protocol (LSP) supports two methods for sending updates: diffs of atomic changes and complete transmission of changed files. The former requires incremental parsing and analysis, which are challenging to implement but enable much faster processing of files upon changes. Incremental parsing relies on an internal representation of the source code that allows efficient updates for small changes. This method necessitates providing the parser with the right context to correctly parse a changed fragment of code.

In practice, most language servers re-index the entire file when changes occur, discarding the previous internal state. This approach is more straightforward to implement as it poses fewer requirements for the architects of the language server, but it is significantly less performant. Unlike incremental processing, which updates only the affected portions of its internal structure, even minor changes such as adding or removing lines require reprocessing the entire file. This method may suffice for small languages and codebases but becomes a performance bottleneck with larger ones.

For code analysis, LSP implementers must choose between lazy and greedy approaches to processing files and answering requests. Greedy implementations resolve most available information during the file's initial indexing, enabling the server to answer requests using simple lookups. Conversely, lazy approaches resolve only minimal local information during indexing, invoking ad-hoc resolution for requests and possibly memoizing the results for future use. Lazy resolution is more common with incremental indexing, as it reduces the work associated with file changes, which is crucial for complex languages that would otherwise require a significant amount of redundant work.

2.2 Language Workbenches

The term *Language Workbench* was coined by Martin Fowler in 2005 [5] to describe a set of tools that support the development of domain-specific languages (DSLs) in the context of *language-oriented programming* paradigm [8].

A language workbench will typically include tools to support the definition, reuse and composition of domain-specific languages together with their integrated development environment.

Current language workbenches usually support:

- Specification of the language concepts or metamodel
- Specification of the editing environments for the domain-specific language
- Specification of the execution semantics, e.g. through interpretation and code generation

Language workbenches have been developed for various technological spaces. Some of the most popular language workbenches are:

- **JetBrains MPS**[11]: A powerful language workbench that allows the creation of domain-specific languages and the generation of code from them.
- **Xtext**[12]: A framework for developing domain-specific languages and languages for code generation.
- **Spoofax**[13]: A language workbench for developing textual domain-specific languages.

Instead, we discuss `Neverlang` (Sect. 2.2.1) in particular detail, since most of the research discussed in this dissertation will use Neverlang as a running example.

2.2.1 Neverlang

The *Neverlang* [3, 4, 7] framework promote the code reusability and the separation of concerns in the implementation of programming languages, based around the concept of *language-feature*. The basic development unit is the **module**, as shown in line 1 of Listing 2.2. A module may contain a **reference syntax** and could have zero or multiple **role**s. A role, used to define the semantics, is unit of composition that defines actions that should be executed when some syntax is recognized, as defined by *syntax-directed translation* [1]. Syntax definitions are defined using *Backus–Naur form* (BNF) grammars, represented as sets of *productions* and *terminals*. Syntax definitions and semantic **role**s are tied together using *slice*s. Listing 2.2 shows a simple example of a Neverlang module implementing a backup task of the `LogLang` LPL. Reference syntax is defined in lines 2-6; the *categories* (line 5) are used also to generate the syntax highlighting for the IDEs. Semantic actions may be attached to a non-terminal using the production's name as a reference, or using the position of the non-terminal in the grammar, as shown in

[11] https://www.jetbrains.com/mps/
[12] https://www.eclipse.org/Xtext/
[13] https://www.metaborg.org/en/latest/

```
1  module Backup {
2    reference syntax {
3      Backup: Backup ⇐ "backup" String String;
4      Cmd: Cmd      ⇐ Backup;
5      categories  : Keyword = { "backup" };
6    }
7    role(execution) {
8      0 .{
9          String src  = $1.string, dest = $2.string;
10         $$FileOp.backup(src, dest);
11     }.
12   }
13 }
14 slice BackupSlice {
15   concrete syntax from Backup
16   module Backup with role execution
17   module BackupPermCheck with role permissions
18 }
19 language LogLang {
20   slices BackupSlice RemoveSlice RenameSlice
21          MergeSlice Task Main LogLangTypes
22   endemic slices FileOpEndemic PermEndemic
23   roles
24     syntax < terminal-evaluation < permissions : execution
25 }
```

Listing 2.2. *Syntax and semantics for the backup task..*

line 8, numbering start with 0 from the top left to the bottom right. The two *String* non-terminals on the right-hand side of the *Backup* production are referenced using 1 and 2, respectively. Different semantic actions may be attached to the same production thanks to the multiplicity of **role**s that can be defined for a module. Each **role** is a compilation phase that can be executed in a specific order, as shown in line 24. In contrast, the `BackupSlice` (lines 14-18) reveals how the syntax and semantics are tied together; choosing the `concrete syntax` from the Backup module (line 15), and two `roles` from two different modules (lines 16-17). Finally, the `language` can be created by composing multiple **slices** (line 20). The composition in Neverlang is twofold: between modules and between slices. Thus, the grammars are merged in order to generate the complete language parser. On the other hand, the semantic actions are composed in a pipeline, and each **role** traverses the syntax tree in the order specified in the **roles** clause (line 24).

2.3 Type Systems

2.4 Software and Language Product Lines

3
Concept

4
Implementation

5
Evaluation

6
Related Work

7 Conclusions

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[4] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[5] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog, May 2005.

[6] 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.

[7] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

[8] Martin P. Ward. Language Oriented Programming. *Software—Concept and Tools*, 15(4):147–161, October 1994.