Chapter 1

Related work

The Nickel Language Server follows a history of previous research and development in the domain of modern language tooling and editor integration. Most importantly, it is part of a growing set of LSP integrations. As such, it is important to get a picture of the field of current LSP projects. This chapter will survey a varied range of popular language servers, compare common capabilities, and implementation approaches. Additionally, this part aims to recognize alternative approaches to the LSP, in the form of legacy protocols, extensible development platforms LSP extensions and the emerging Language Server Index Format.

1.1 Language Servers

1.1.1 Considerable dimensions

1.1.1.1 Language Complexity

1.1.1.2 LSP compliance

1.1.1.3 Features

1.1.1.4 File processing

1.1.1.4.1 Incremental

1.1.1.4.2 Full

1.1.2 Comparative Projects

1.1.3 Honorable mentions

1.2 Alternative approaches

1.2.1 LSP Extensions

The LSP defines a large range of commands and capabilities which is continuously being extended by the maintainers of the protocol. Yet, occasionally server

developers find themselves in need of functionality not yet present in the protocol. For example the LSP does not provide commands to exchange binary data such as files. In sec. **??** the CodeCompass Language Server was introduced. A stern feature of this server is the ability to generate and show diagrams in SVG format. However, the LSP does not define the concept of *requesting diagrams*. In particular Mészáros et al. describe different shortcomings of the LSP :

1. "LSP doesn't have a feature to place a context menu at an arbitrary spot in the document" Context menu entries are implemented by clients based on the agreed upon capabilities of the server. Undefined capabilities cannot be added to the context menu.

   In the case of CodeCompass the developers made up for this by using the code completion feature as an alternative dynamic context menu.

2. "LSP does not support displaying pictures (diagrams)." CodeCompass generates diagrams for selected code. Yet, there is no image transfer included with the LSP. Since the LSP is based on JSON-RPC messages, the authors' solution was to define a new command, specifically designed to tackle this non-standard use case.

Missing features of the protocol such as the ones pointed out by Mészáros et al. appear frequently, especially in complex language servers or ones that implement more than basic code processing.

The rust-analyzer defines almost thirty non-standard commands (**rust-analyzer-extensions?**), to enable different language specific actions.

Taking the idea of the CodeCompass project further, Rodriguez-Echeverria et al. propose a generic extension of the LSP for graphical modeling (**lsp-for-graphical-modeling?**). Their approach is based on a generic intermediate representation of graphs which can be generated by language servers and turned into a graphical representation by the client implementation.

Similarly, in (**decoupling-core-analysis-support?**) the authors describe a method to develop language agnostic LSP extensions. In their work they defined a language server protocol for specification languages (SLSP) which builds on top of the existing LSP, but adds several additional commands. The commands are grouped by their use case in the domain of specification languages and handled by separate modules of the client extension implementing support for the SLSP. Following the LSP example and defining commands language agnostic instead of tied to a specific language, allows to maintain the initial purpose of the LSP. Since the extensions can be incorporated by specific implementations of language servers in the same domain, a single client implementation serves multiple languages. The authors point out that while their approach specializes in specification languages, the idea can be transferred to other areas.

1.2.2 Language Server Index Format

The Language Server Index Format, or short LSIF, is an augmentation of the LSP. Since the LSP requires a language server to actively analyze files and answer requests, its use is typically constrained to local installations. As pointed out

in the introducing blog post (**lsif-blog-post?**), several use cases exist where the LSP approach fails due to resource limits. The article explicitly names web based platforms such as GitHub(**github?**) or Sourcegraph(**sourcegraph?**).

The LSIF aims to provide the features of the LSP without the need of actively running a language server. Instead, "language servers or other programming tools to emit their knowledge about a code workspace" as a LSIF compliant JSON report.

The LSIF specification (**lsif-spec?**) defines four principal goals:

- The format should not imply the use of a certain persistence technology.
- The data defined should be modeled as closely as possible to the Language Server Protocol to make it possible to serve the data through the LSP without further transformation.
- The data stored is result data usually returned from an LSP request.
- The output format will be based on JSON as with the LSP.

The format specifies a graph structure that comprises that links ranges of source code to language analysis results that are based on the data types defined by the LSP. Vertices represent higher level concepts such as `document`s, `range`s, `resultSet`s and actual results. The relation between vertices is expressed through the edges.

Referring to an example form the official specification (**lsif-spec?**), an analysis of the code sample in lst. **??** may produce hover information for the function `bar()`. Using the LSIF, the result would be encoded as seen in lst. **??**. The graph structure encoded here is visualized in fig. **??**. Using this graph an LSIF tool is able to resolve statically determined hover information by performing the following steps.

1. search for `textDocument/hover` edges
2. select the edge that originates at a `range` vertex corresponding to the requested position.
3. return the target vertex

As a consequence, a subset of LSP capabilities can be provided statically based on the preprocessed LSIF model.

1.2.3 *SP, Abstracting software development processes

Since its introduction the Language Server Protocol has become a standard format to provide language tooling for editing source code. Meanwhile, as hinted in sec. **??**, the LSP is not able to fully satisfy every use-case sparking the development of various LSP extensions. Following the success of language servers, similar advances have been made in other parts of the software development process.

For instance, many Java build tools expose software build abstractions through the Build Server Protocol (**build-server-protocol?**), allowing IDEs to integrate more languages more easily by leveraging the same principle as the LSP. The BSP provides abstractions over dependencies, build targets, compilation and running of projects. While the LSP provides `run` or `test` integration for selected

languages through Code Lenses, this is not part of the intended responsibilities of the protocol. In contrast, those tasks are explicitly targeted by the BSP.

Next to *writing* software (LSP) and *building/running/testing* software (e.g. BSP), *debugging* presents a third principal task of software development. Similar to the other tasks, most actions and user interfaces related to debugging are common among different languages (stepping in/out of functions, pausing/continuing exection, breakpoints, etc.). Hence, the Debug Adapter Protocol, as maintained by Microsoft and implemented in the VSCode Editor, aims to separate the language specific implementation of debuggers from the UI integration. Following the idea of the LSP, the DAP specifies a communication format between debuggers and editors. Since debuggers are fairly complicated software, the integration of editor communication should not prompt new developments of debuggers. Instead, the DAP assumes a possible intermediate debugger adapter do perform and interface with existing debuggers such as `LLDB`, `GDB`, `node-debug` and others(**DAP-impls?**).

Following the named protocols, Jeanjean et al. envision a future (**reifying?**) where all kinds of software tools are developed as protocol based services independent of and shared by different IDEs and Editors. Taking this idea further, they call for a Protocol Specification that allows to describe language protocols on a higher level. Such a protocol, they claim, could enable editor maintainers to implement protocol clients more easily by utilizing automated generation from Language Service Protocol Specifications. Additionally, it could allow different Language Services to interact with and depend on other services.