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

The LSP project was announced (**lsp-announced?**) in 2016 to establish a common protocol over which language tooling could communicate with editors. The LSP helps the language intelligence tooling to fully concentrate on source analysis instead of integration with specific editors by building custom GUI elements and being restricted to editors extension interface.

At the time of writing the LSP is available in version `3.16` (**Spec?**). Microsoft's official website lists 172 implementations of the LSP(**implementations?**) for an equally impressive number of languages.

An in-depth survey of these is outside the scope of this work. Yet, a few implementations stand out due to their sophisticate architecture, features, popularity or closeness to the presented work.

1.1.1 Considerable dimensions

To be able to compare and describe LSP projects objectively and comprehensively, the focus will be on the following dimensions.

Target Language The complexity of implementing language servers is influenced severely by the targeted language. Feature rich languages naturally require more sophisticated solutions. Yet, existing tooling can often be leveraged to facilitate language servers.

Features The LSP defines an extensive array of capabilities. The implementation of these protocol features is optional and servers and clients are able to communicate a set of *mutually supported* capabilities.

The Langserver (**langserver.org?**) project identified six basic capabilities that are most widely supported:

1. Code completion,
2. Hover information,
3. Jump to definition,
4. Find references,
5. Workspace symbols,
6. Diagnostics

Yet, not all of these are applicable in every case and some LSP implementations reach for a much more complete coverage of the protocol.

File Processing Most language servers handling source code analysis in different ways. The complexity of the language can be a main influence for the choice of the approach. Distinctions appear in the way servers process *file indexes and changes* and how they respond to *requests*.

The LSP supports sending updates in form of diffs of atomic changes and complete transmission of changed files. The former requires incremental parsing and analysis, which are challenging to implement but make processing files much faster upon changes. An incremental approach makes use of an internal representation of the source code that allows efficient updates upon small changes to the source file.

Additionally, to facilitate the parsing, an incremental approach must be able to provide a parser with the right context to correctly parse a changed fragment of code. In practice, most language servers process file changes by re-indexing the entire file, discarding the previous internal state entirely. This is a more approachable method, as it poses less requirements to the architects of the language server. Yet, it is far less performant. Unlike incremental processing (which updates only the affected portion of its internal structure), the smallest changes, including adding or removing lines effect the *reprocessing of the entire file*. While sufficient for small languages and codebases, non-incremental processing quickly becomes a performance bottleneck.

For code analysis LSP implementers have to decide between *lazy* or *greedy* approaches for processing files and answering requests. Dominantly greedy implementations resolve most available information during the indexing of the file. The server can then utilize this model to answer requests using mere lookups. This stands in contrast to lazy approaches where only minimal local information is resolved during the indexing. Requests invoke an ad-hoc resolution the results of which may be memoized for future requests. Lazy resolution is more prevalent in conjunction with incremental indexing, since it further reduces the work associated with file changes. This is essential in complex languages that would otherwise perform a great amount of redundant work.

1.1.2 Representative LSP Projects

Since the number of implementations of the LSP is continuously growing, this thesis will present a selected set of notable projects.

1. Three highly advanced and complete implementations that are the de-facto standard tooling for their respective language: *rust-analyzer* (**rust-analyzer?**), *ocaml-lsp/merlin* (**merlin?**) and the *Haskell Language Server* (**hls?**)
2. Two projects that provide compelling alternatives for existing specialized solutions: *Metals* (for Scala) (**metals?**), *Java LSP* (**java-lsp?**)
3. Language Servers for small languages in terms of complexity and userbase, highlighting one of the many use cases for the LSP. *rnix-lsp* (**rnix-lsp?**), *frege-lsp*

1.1.2.1 LSP as standard tooling

Today LSP-based solutions serve as the go-to method to implement language analysis tools. Emerging languages in particular take advantage from the the flexibility and reach of the LSP.

Most languages profit greatly from the possibility to leverage existing tooling for the language. For instance the Haskell language server facilitates a plugin system that allows it to integrate with existing tooling projects (**hls-plugin-search?**). Plugins provide capabilities for linting (**hls-hlint-plugin?**), formatting (**hls-floskell-plugin?**,hls-ormolu-plugin), documentation (**hls-haddock-comments-plugin?**) and other code actions (**hls-tactics-plugin?**) across multiple compiler versions. While this requires HSL to be compiled with the same compiler version in use by the IDE, it also avoids large scale reimplementations of compiler features in an entirely different language.

Similarly, the Ocaml language service builds on top of existing infrastructure by relying on the Merlin project introduced in sec. **??**. Here, the advantages of employing existing language components have been explored even before the LSP.

The rust-analyzer (**rust-analyzer?**) takes an intermediate approach. It does not reuse or modify the existing compiler, but instead implements analysis functionality based on low level components. This way the developers of rust-analyzer have greater freedom to adapt for more advanced features. For instance rust-analyzer implements an analysis optimized parser with support for incremental processing. Due to the complexity of the language, LSP requests are processed lazily, with support for caching to ensure performance. While many parts of the language have been reimplemented with a language-server-context in mind, the analyzer did not however implement detailed linting or the rust-specific borrow checker. For these kinds of analysis, rust-analyzer falls back to calls to the rust build system.

1.1.2.2 Rapid implementation of LSP features

The LSP makes it possible to bring language support to many editors at once. Emerging languages are profiting from this in particular since development resources are typically scarce and the core is quite volatile. To mitigate this,

core language components can often be reused or integrated through command line calls.

A good example is given by the Frege language (**frege-github?**). Frege as introduced in (**frege-paper?**) is a JVM based functional language with a Haskell-like syntax. It features lazy evaluation, user-definable operators, type classes and integration with Java.
While previously providing an eclipse plugin (**frege-eclipse?**), the tooling efforts have since been put towards an LSP implementation. The initial development of this language server has been reported on in (**frege-lsp-report?**). The author shows though multiple increments how they utilized the JVM to implement a language server in Java for the (JVM based) Frege language. In the final proof-of-concept, the authors build a minimal language server through the use of Frege's existing REPL and interpreter modules. Being written in Java, allows the server to make use of other community efforts such as the LSP4J project which provide abstractions over the interaction with LSP clients. Through the use for such abstractions, servers can focus on the implementation of capabilities only.

1.1.2.3 Language Independence

The LSP is truly language independent. Many language implementations do not expose the required language interfaces (parsing, AST, Types, etc..), or pose various other impediments such as a closed source, licensing, or the absence of LSP abstractions available for the host language.

An instance of this type is the rnix-lsp(**rnix-git?**) language server for the nix(**nixos.org?**) programming language. Despite the nix language being written in C++ (**nix-repo?**), its language server builds on a custom parser(**rnix?**) in Rust.

The work presented by Leimeister in (**cpachecker-lsp?**) exemplifies how functionality can be provided by entirely external tools as well. The server can be used to automatically perform software verification in the background using CPAchecker(**cpachecker?**). CPAchecker is a platform for automatic and extensible software verification. The program is written in Java and provides a command line interface to be run locally. Additionally, it is possible to execute resource intensive verification through an HTTP-API on more powerful machines or clusters (**cpa-clusters?**). The LSP-Server supports both modes of operation. On one hand it interfaces directly with the Java modules provided by the CPAchecker library, on the other it is able to utilize an HTTP-API provided by a server instance of the verifier.

Similar to the work by Leimeister, in (**comprehension-features?**) Mészáros et al. present a proof of concept leveraging the LSP to integrate (stand-alone) code comprehension tools with the LSP compliant VSCode editor. Code comprehension tools support the work with complex code bases by "providing various textual information, visualization views and source code metrics on multiple abstraction levels." Pushing the boundaries of LSP use-cases, code comprehension tools do not only analyze specific source code, but also take into account contextual information. One of such tools is CodeCompass (**code-compass?**). The works of Mészáros yielded a language server that allowed to access the analysis

features of CodeCompass in VSCode. In their paper they specifically describe the generation of source code diagrams. Commands issued by the client are processed by a CodeCompass plugin which acts as an LSP server and interacts with CodeCompass through internal APIs.

1.1.2.4 Language Servers for Domain Specific Languages

Bünder and Kuchen (**multi-editor-support?**) highlight the importance of the LSP in the area of Domain Specific Languages (DSL). Compared to general purpose languages, DSLs often targets both technical and non-technical users. While DSL creation workbenches like Xtext (**eclipse-xtext?**), Spoofax (**spoofax?**) or MPS(**jetpbrains-mps?**) allow for the implementation and provision of Eclipse or IntelliJ based DSLs, tooling for these languages is usually tied to the underlying platform. Requiring a specific development platform does not satisfy every user of the language. Developers have their editor of choice, that they don't easily give up on. Non-technical users could easily be overwhelmed by a complex software like Eclipse. For those non-technical users, a light editor would be more adapted, or even one that is directly integrated into their business application. The authors of (**multi-editor-support?**) present how Xtext can generate an LSP server for a custom DSL, providing multi-editor support. The authors especially mention the Monaco Editor (**monaco-editor?**), a reusable HTML component for code editing using web technologies. It is used in products like VSCode (**vscode?**), Theia (**theia?**) and other web accessible code editors. The Monaco Editor supports the LSP as a client (that is, on the editor side). Such LSP-capable web editors make integrating DSLs directly into web applications easier than ever before.

1.1.3 Honorable mentions

1.2 Alternative approaches

1.2.1 Platform plugins

1.2.2 Legacy protocols

1.2.3 LSP Extensions

1.2.4 LSIF