MCIT Modeling, control & information technologies

The Case for Asynchronous Language Support API in an Integrated Development Environment

https://doi.org/10.31713/MCIT.2024.038

Yurii Turbal
National University of Water and Env. Engineering
Rivne, Ukraine
turbaly@gmail.com

Igor Kushnir
National University of Water and Env. Engineering
Rivne, Ukraine
igorkuo@gmail.com

Abstract—**This paper explains why adding Language Server Protocol support to an existing integrated development environment requires replacing its synchronous language support API with an asynchronous one. We explore the possibility of reusing language support code of other applications during the transition to the asynchronous API. And finally, we envision substantial additional benefits of such a transition.**

Keywords—integrated development environment; Language Server Protocol; asynchronous API

I. Introduction

Software engineers rely heavily on their integrated development environments (IDEs). Functionality, convenience, and performance of an IDE can have significant impact on an engineer's productivity, speed and quality of software development.

Most modern IDEs are large and complex applications. They are often developed for decades by software engineering teams that change over time. A new IDE must overcome high barriers of entry to compete with existing applications and to become widely used.

Perhaps the most important and difficult problem of IDE development is programming language support. Saving effort on this problem is paramount for an IDE that supports multiple programming languages, especially when the languages evolve over time.

Language Server Protocol (LSP) is currently without alternatives in terms of the number of programming languages that can be supported with little effort. Compared to the code bases of most popular IDEs, the protocol is very young – its standardization process started only in 2016. But the protocol's success is tremendous: in the early 2020s the LSP became a de-facto standard in the IDE market [1].

KDevelop is a well-established free/libre, cross-platform IDE. This IDE currently supports four programming languages well: C++, C, Python, and PHP. KDevelop's language support architecture dates back to 2007 and is not compatible with the much newer LSP standard.

II. Related Work

Paper [2] aims to reduce the effort required to implement IDE support for less popular programming languages and proposes a new parse-based design for language servers. Paper [3] analyzes the implementations of existing language servers and synthesizes implementation practices. Paper [4] describes writing a new IDE from scratch using the LSP.

However, the problem of integrating the LSP into an existing IDE is insufficiently researched and documented. This paper aims to fill the gap. Improving an existing IDE circumvents the high barriers of entry and has an immediate positive impact on the productivity of software engineers that use it.

III. Overview of the Language Server Protocol

In order to perform its functions, e.g. syntax highlighting, code completion, navigation, an IDE must "understand" programming languages. Traditionally each IDE implemented language support separately and independently from other IDEs. At best, an IDE provided a custom application programming interface (API) to language module (plugin) developers. The work required to support n languages in m IDEs was then $O(n \cdot m)$. Popular programming languages are never complete, new standard versions are released regularly. Therefore, language support code has to be regularly revised and updated. In practice, this meant that an IDE supported well at most a tiny number of languages.

The idea behind a language server is to encapsulate knowledge of a programming language inside a server that can communicate with development tools, such as IDEs, over a protocol that enables inter-process communication. The idea behind the Language Server Protocol is to standardize the protocol for how tools and servers communicate, so that a single language server can be reused in multiple development tools, and tools can support languages with minimal effort [5]. This reduces the work required to support n languages in m tools (IDEs) to $O(n+m)$.

A language server runs as a separate process and development tools communicate with the server using the language protocol over JSON-RPC. An example for how a tool and a language server communicate during a routine editing session is displayed on Fig. 1.

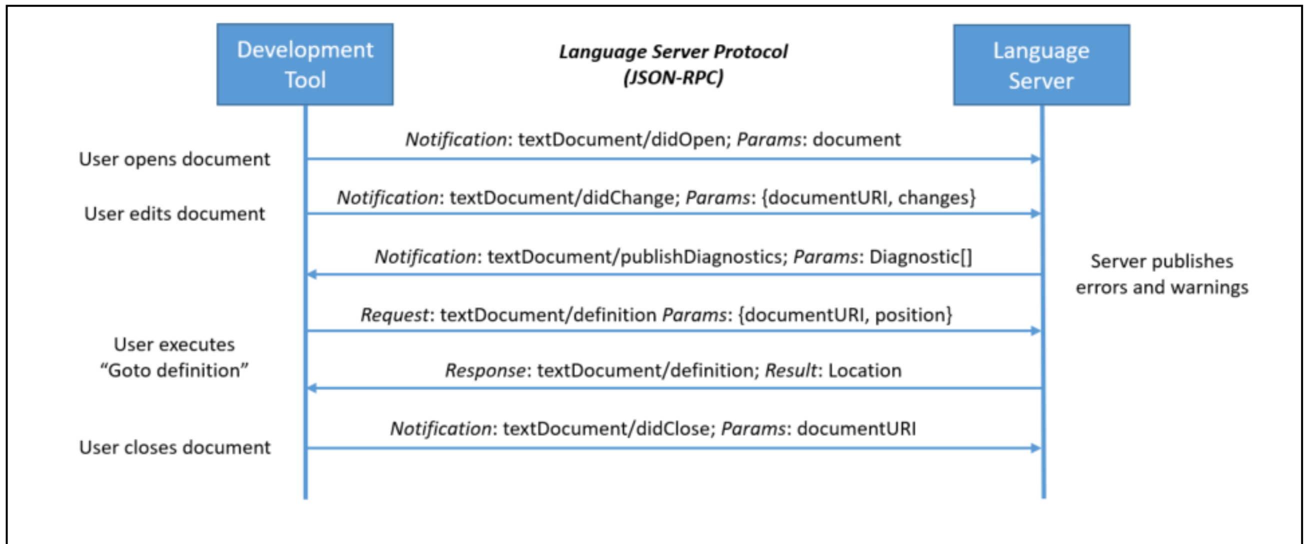

Figure 1. Example of LSP client-server communication [5]

IV. Overview of the Duchain Framework

"Duchain" is the name of a long-standing language support framework of the KDevelop IDE. The name of the framework is derived from a data structure that represents a source code file. The Duchain documentation [6] describes the framework in detail and is heavily quoted in the next four paragraphs.

The Definition-Use Chain (abbreviated as duchain) is a sequence of contexts in a source code file, and the associated definitions which occur in those contexts. A simplified way of thinking about it is that for each set of brackets (curly *{}* or not *()*), there is a separate context. Each context is represented by an object of the type *KDevelop::DUContext*. Each context has a single parent context (except for the top-level contexts, which have none), and any number of child contexts (including none). Additionally, each context can import any number of other contexts. Thus, the *KDevelop::DUContext* structure resembles a directed acyclic graph.

These *DUContext* objects are created during the first pass after parsing the code to an abstract syntax tree (AST). Also, at this stage the data types are parsed, and any declarations, which are encountered, are recorded against the context in which they are encountered in. Each declaration is represented by a *KDevelop::Declaration* object.

Creating a definition-use chain for a programming language requires implementing the following:

- a parser for the language,
- a context builder,
- a type builder,
- a declaration builder,
- a use builder.

Code completion support requires further work specific to the programming language.

Numerous duchain-based plugins have been implemented to let KDevelop support different programming languages. Unfortunately, most of these plugins have never been completed. Even some completed plugins are not regularly maintained and eventually stop working in latest KDevelop version.

KDevelop is primarily a C++ IDE, so its C++ language support plugin traditionally benefits from abundant developer attention. After all, KDevelop is written in C++, and its code is usually developed in KDevelop itself. The first such plugin had supported C++ language features heuristically. Since 2011, a new C++ language standard is published every three years. The rapid language evolution makes supporting all the new features heuristically a maintenance nightmare. Because of that, a new plugin was developed based on libclang. Libclang is the C Interface to one of the few major C++ compilers – Clang. Eventually, the old heuristic plugin was removed as rarely used and no longer maintained.

V. Integrating the Language Server Protocol into KDevelop

Both C++ plugins and all other KDevelop language plugins are based on the Duchain framework. Once built, a definition-use chain can be examined and navigated using a synchronous API (without callbacks). Numerous high-level IDE features, such as code completion, semantic code highlighting, context browsing, identifier details and documentation, quick open, are implemented in terms of the long-lived duchain API and depend heavily on its synchronous nature.

A definition-use chain is built from an abstract syntax tree and contains a sequence of contexts that represent regions of code. An LSP client cannot possibly create a duchain, because the LSP API is much higher-level than that. The closest that an LSP server reply can offer is a list of symbols (the Document Symbols Request), that is, a list of variables, types, functions, numbers, etc. [7]

An LSP server resides in a separate process, so an LSP client cannot implement a synchronous API

required by high-level KDevelop features efficiently. A new asynchronous API has to be designed to properly integrate the LSP into KDevelop. The high-level IDE features must be ported to the asynchronous API. And the asynchronous API must be implemented using the existing synchronous duchain API to keep existing language plugins working. Finally, a new plugin that implements the asynchronous API using the LSP can be developed.

Such a proper integration of the LSP would vastly decrease the effort required to add new language support to KDevelop and maintain it over the long term. The LSP language support would likely be more complete compared to rarely maintained duchain-based plugins.

The LSP is not perfect or complete, but evolves over time. Integrating the LSP into yet another feature-rich IDE is bound to reveal functionality gaps or defects in existing features of the protocol. This can lead to improvements of the LSP standard and benefit all IDEs that use it.

VI. Possible Reuse of Existing Language Support Interfaces and Implementations

The implementation of KDevelop is based on the Qt application development framework and on the KDE Frameworks (a set of Qt add-on libraries). In particular, many of KDevelop's syntax highlighting and text editing features are implemented by the KSyntaxHighlighting and KTextEditor KDE frameworks. A free/libre, cross-platform text editor application Kate is also based on these two frameworks and is maintained by the frameworks' developers themselves. An LSP client plugin for Kate was first released in 2019 and keeps evolving [8]. However, programming language support is a central feature of an IDE but only an optional add-on feature of a text editor. Therefore, the LSP should be integrated much more tightly in KDevelop than in Kate. Still, Kate's LSP client plugin can be used as a solid starting point of the future LSP plugin for KDevelop, because the two code bases are closely related.

Qt Creator is another free/libre, cross-platform, primarily C++ IDE. It is implemented using the same application development framework as KDevelop – Qt, though it does not depend on the KDE frameworks. Therefore, analyzing Qt Creator's approaches can offer insights into language support API redesign for KDevelop; parts of Qt Creator's LSP client implementation can potentially be reused.

Qt Creator had traditionally supported C++ language features heuristically, similarly to how KDevelop was doing it before the libclang-based plugin. Then a clangd Qt Creator plugin arrived. Clangd is a C++ language server based on the Clang C++ compiler. Clangd's C++ language support is much more complete than that of the heuristic parser. But Qt Creator still offers an option to disable the clangd plugin and use the old parser instead, because it uses much less random access memory (RAM). Thus Qt Creator supports two very different C++ language support backends. We aim to achieve a similar result in KDevelop for C++ and other languages: support both the duchain and the LSP backends. Qt Creator also offers specialized support for Python, QML, and Java language servers, which proves that its language support interfaces are not limited to C++.

An example of a language support interface specific to C++ is Qt Creator's *CppEditor::ModelManagerSupport* abstract class, which offers API such as *void followSymbol(const CursorInEditor &data, const Utils::LinkHandler &processLinkCallback, FollowSymbolMode mode, bool resolveTarget, bool inNextSplit)* and *void findUsages(const CursorInEditor &data) const*. The interface is inherited by two classes *BuiltinModelManagerSupport* (heuristic) and *ClangModelManagerSupport* (clangd). The interface is necessarily asynchronous, because the clangd language server lives in a separate process. Therefore, the API member functions return *void*, and *std::function* callbacks are used to allow asynchronous implementations reply to requests in their own time.

VII. Other Benefits of Asynchronous Language Support API

Besides enabling the LSP integration, asynchronous language support API also helps to address other long-standing IDE problems. Two such problems that affect KDevelop are described below.

A synchronous call to a language-support library's (e.g. libclang's) function or to a higher-level IDE function can take a long time and freeze the user interface (UI) for seconds. This annoys, distracts, and demotivates software engineers that use the IDE. Asynchronous API allows improving UI responsiveness by performing the most time-consuming work in separate threads or even separate processes, and leaving the main UI thread free to handle user interactions.

Bugs in language-support libraries can make them crash. When core, language support, and UI code of an IDE resides in a single common process, such a crash brings down the entire IDE. Fixing bugs like this in huge language-support libraries, such as libclang, can be very time-consuming, especially for IDE developers who are not well-versed in the implementation details of the libraries. When language support code, including an external library, resides in a separate process, only that process alone crashes. Then the main IDE process can restart the language-support process, and the user would experience only the minor inconvenience of temporary interruption of language-related updates, or possibly not even notice the crash at all. Asynchronous API is the necessary first step in moving language support code into a separate process.

Acknowledgment

Special thanks to KDevelop developers Milian Wolff and Sven Brauch for ideas about how to properly integrate the LSP into KDevelop.

References

[1] D. Bork and P. Langer, "Catchword: Language Server Protocol: an introduction to the protocol, its use, and adoption for web modeling tools," Enterprise Modelling and Information Systems Architectures, vol. 18, no. 9, pp. 1–16, 2023.

[2] S. Marr, H. Burchell, and F. Niephaus, “Execution vs. parse-based language servers: tradeoffs and opportunities for language-agnostic tooling for dynamic languages,” in Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages (DLS ’22), December 07, 2022, Auckland, New Zealand, pp. 1-14.

[3] D. Barros, S. Peldszus, W. K. G. Assunção, and T. Berger, “Editing support for software languages: implementation practices in language server protocols,” in ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS ’22), October 23–28, 2022, Montreal, QC, Canada. ACM, New York, NY, USA, 12 pages.

[4] N. Mitchell, M. Kiefer, P. Iborra, et al., “Building an integrated development environment (IDE) on top of a build system: the tale of a Haskell IDE,” conference: IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages.

[5] The LSP overview. URL: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/

[6] The Duchain framework documentation. URL: https://commits.kde.org/kdevelop?path=kdevplatform/language/duchain/Mainpage.dox

[7] The LSP specification. URL: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[8] The LSP Client Plugin for Kate documentation. URL: https://docs.kde.org/stable5/en/kate/kate/kate-application-plugin-lspclient.html