

Towards a Resident Static Analysis

Maxim Menshikov(✉)

Saint Petersburg State University, 7-9, Universitetskaya nab,
St. Petersburg 199034, Russian Federation
info@menshikov.org

Abstract. The software engineering industry is moving from run-once tools to resident servers. This is well seen in the compiler industry, which accelerated the development of Language Server protocols used in Integrated Development Environments. Static analysis can also benefit from it. In our static analyzer framework, we experimented with making the analysis resident. The implementation comprises the mediator serving user's requests and analyzer units doing static analysis, saving the results to shared or local databases. Thus, the resident analyzer is an adaptation of a software-as-a-service model to the field. In the paper, we present the stages the analyzer goes through, including controlled compilation, data fetching, preliminary and on-demand analysis, show their corner cases. The model was tested in two scenarios. For a single query, the resident analyzer is acting 6–40% worse than a standalone version with a defined analysis plan, which is expected due to memory usage optimization. In a failover scenario, the rescheduling avoids data loss at the cost of increased computation time. These effects shape the differences between models and their applications.

Keywords: Resident server · Static analysis · Architecture · Software-as-a-service

1 Introduction

The industrial software is moving towards a service model. The compilers are evolving into compiler servers powered by Language Server Protocols, which are used by a variety of IDEs, linters, and analyzers. Static analyzers like Coverity [2], SonarQube/SonarCloud[1] are making such model available through the cloud, via continuous integration services. During the evolution of our static analysis framework, briefly mentioned in [8], we observed the need to bring the analyzer to an interactive level as spanning a new analyzer instance every time is resource-consuming. When checking tools available commercially, we noticed that essentially no analyzer performs deep analysis in real time, with all the architecture available on demand.

[1] https://sonarcloud.io.

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11620, pp. 62–71, 2019.
https://doi.org/10.1007/978-3-030-24296-1_7

This paper summarizes the author's approach to making an analyzer resident.

Goal. Turn the static analysis framework into a resident service. The main contribution of the paper is a model for performing such an analysis type.

Motivation. The need to add interactivity to analysis, enable querying of program properties, and the essence of building it the proper way.

Novelty. According to the observations described in Sect. 2, the idea to keep analysis resident as opposed to IDE-bound or standalone model is new (at least authors are not aware of the analogs), therefore the architecture is novel.

Methodology. We employ engineering methods widespread in the industry. The main contribution is a result of task decomposition combined with the distributed technology stack and a critical view on the analyzer's workflow.

The paper is organized as follows. In Sect. 2 we review existing solutions. In Sects. 3 and 4 we propose and describe our approach. The implementation of the approach is evaluated in Sect. 5 and discussed in Sect. 6.

2 Related Work

SonarQube [3] is close to the idea. We observe a few substantial differences from our idea: the project is focused on non-interactive problem search (although SonarLint can be used for real-time feedback). Also, the SonarQube saves results of analysis rather than semantic data, which is particularly useful for tackling memory issues, but ultimately prohibits advanced queries.

Language Server Protocol [1] is Microsoft's effort to unite Integrated Development Environments with compilers and development tools. It sparked a number of tools and servers, such as Clangd[2], Rust Language Server[3] and a number of other utilities. It is mainly created for IDEs, so a server is relatively short-lived. Our project is aimed at a longer lifetime.

GrammaTech has announced a Static Analysis Server Protocol[4], which is not yet backed by a specification. It aims at supporting the Static Analysis Result Interchange Format (SARIF).

ReSharper [6] is a tool developed by JetBrains acting as a Visual Studio extension for a deeper analysis of C++/C# source codes. Visual Studio historically had several limitations, such as the 32-bit constraint. This induced major performance and memory issues, which were overcome by moving the tool out of process[5]. ReSharper is proprietary, so it is hard to judge about its internal

[2] https://clang.llvm.org/extra/clangd.html.
[3] https://github.com/rust-lang/rls.
[4] https://resources.grammatech.com/blog-3/static-analysis-results-a-format-and-a-protocol-sarif-sasp.
[5] https://blog.jetbrains.com/dotnet/2018/05/29/taking-resharper-process-resharper-performance-series.

architecture, however, an opensource intercommunication protocol[6] used in the tool is worth investigation.

Overall, no solution allows queries on the source code while keeping the semantic data cached.

3 Proposed Method

The approach is a direct evolution of a Semantic Virtual Machine framework [8]. However, hypervisor-based load distribution and virtual machine code execution model don't really handle user's requests. With this paper, we are placing the hypervisor at the level of individual executors. Before discussing technical details, let's compare different models.

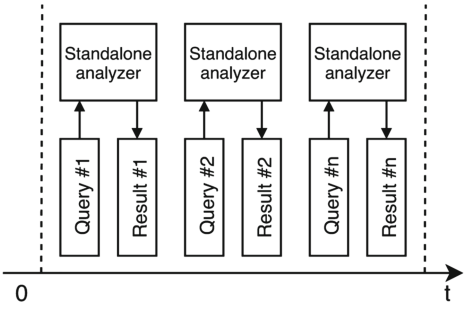

Fig. 1. Standalone model example

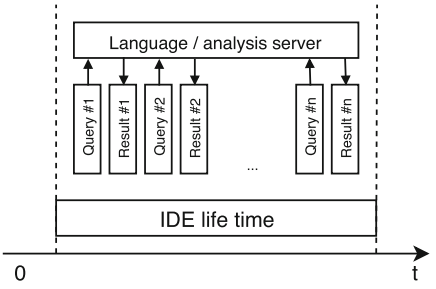

Fig. 2. Language/analysis server model example

Analyzer Life Cycle. A static analyzer may run in multiple ways. The first model is *standalone* (Fig. 1). The standalone model implies invoking analysis again for each query. This is redundant because queries are usually run on more

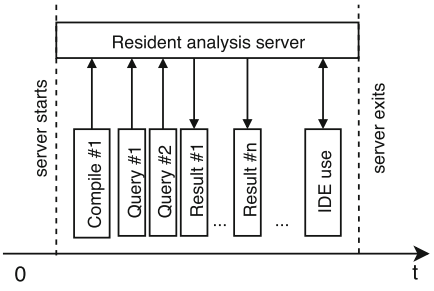

Fig. 3. Resident server model example

[6] https://github.com/JetBrains/rd.

or less the same code base, and sophisticated incremental analysis algorithm is required to make it effective. *Language/analysis server*-like model (Fig. 2) involves running analyzer supplementary to client-side application or IDE. A tool runs in the background, caches results and semantic data. It is bound to IDE and therefore other applications may not be able to take advantage of it. *Resident server* (Fig. 3) can run supplementary, ideologically similar to Software-as-a-Service (SaaS) [5]. This offers a performance improvement for a series of interactive queries made within a long time frame (e.g. within a week).

Load Distribution. The solution is aimed at a distribution of user requests. For small programs, the need for this is not obvious, but as the program grows, the more variables and functions spawn more interconnections. The state space explodes, increasing the memory and CPU requirements. Another problem is the questionable stability of a single analyzer: it is usually not hard to crash or invoke undesired behavior by passing invalid input. The only possible solution is to offload the requests to various units and collect their findings in one place.

Mediator. A *mediator* is put atop of hierarchy. From the end user's point of view, it is simply a server of a client/server architecture. The difference is caused by increased complexity. The mediator *potentially* can execute simple queries, but if query goes beyond a general knowledge of a program structure, the query is sent to units with more specific knowledge of the domain.

Consider the following examples in the pseudo query language.

```
Query 1:
in resource (type = function, name = X), verify \result = 0
```

This query is specific to a single function, it is sent to a unit containing its intermediate representation.

```
Query 2:
in resource (type = function), verify \valid(\result)
```

Whenever the request covers the unit's zone of responsibility, the unit is invoked. This particular request may potentially invoke all units.

Failover. If a unit fails—due to crash, host shutdown or Out Of Memory condition—the engine tries to restore it, and if it is unsuccessful, a job to a different unit is scheduled. In that case, there is a high probability that analyzer would have to verify an apparently missing part of the code and invalidate global database cache. There are simple guarantees of integrity. If a source file is changed, the Invalidation Request must be passed to Mediator, and then the database and all its dependencies (even in other units) are invalidated. Each unit contains a workspace version number, and databases still contain previous versions until all units swap to this version. Thus ***the constant changes of many source files with inherent cross-invalidation do not correspond to a primary use case for the model***.

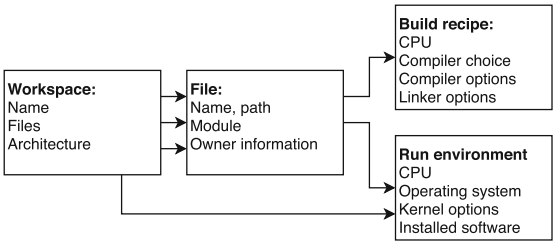

Fig. 4. Workspace schema

The critical part of the failover mechanism is similar to the well-known Google's MapReduce [4] pattern, which adds up the confidence that the model is tested on high-load servers.

Communication Format. The communication message format selection is a debatable question because its choice depends on the environment. Among possible alternatives, we see JSON-RPC—which is free-form and human-readable. It fits the position of the frontend message format. For internal inter-unit/mediator messaging, the Protocol Buffers[7] are our choice.

4 Operating Modes

The whole operation can be divided into 5 core phases. They relate to the creation of workspace, the retrieval of compilation information or sources, analysis of different kinds.

4.1 Workspace Creation

Each workspace is created on a per-request basis. The client submits a Workspace Creation request and obtains a Workspace ID. During this early exchange, the parties use Diffie-Hellman exchange to generate session keys. As a workspace is a permanent entity, unlike a relatively short-lived session, the user might set a secret to access it in the future.

The internals of the workspace and its relations to other entities are presented in Fig. 4. The user is obliged to fill the target environment data as static analyzers depend on it.

4.2 Controlled Compilation

When the analysis server is running for the first time, it might not know anything about the project being analyzed. The client-side agent has to collect the compilation information. A typical approach to solve this problem is to run

[7] https://developers.google.com/protocol-buffers.

agent hand-to-hand to original compiler[8] [2] (or rather *instead* of a compiler) and propagate its arguments.

Our approach is based on it as well, except that gathered arguments are not analyzed on the client and are passed to the server *as is*. The process can be divided into the following phases:

1. Obtaining a build recipe comprising target CPU, compiler options and linker flags.
2. Retrieving owner information from the code repository.
3. Creating a dependency graph between files and modules.
4. Saving the retrieved data to a workspace.

4.3 Data Fetching

The data is obtainable by a variety of ways:

1. If the analyzer is running on the same machine as the user, it can be read as is.
2. The file can be passed by an encrypted channel along with compilation request.
3. The file can be obtained from FTP, taken by SSH, or transparently fetched using NFS.

The client and server must negotiate the ways to deliver information. If the user didn't set up any means, the only reasonable way is using an encrypted channel to pass the whole file. However, as data can be sensitive, the system administrator is obliged to set up methods to avoid a possible data leak channel.

4.4 Preliminary Analysis

Once data is fetched, it can finally be analyzed. In the case of our project, the analysis converts the code to intermediate representation [8], obtains function summaries and saves them up in a database. The stage is triggered automatically after fetching data. The ways to analyze code are detailed sufficiently in other research papers aiming specifically at static analysis.

4.5 Analysis on Demand

This stage is triggered by the user's query. The incoming request contains a file name(s) if selected, and the query expression. The analyzer may either perform a query on one file or the whole workspace (thus requiring a mediator to select a unit to solve the query). The request life cycle is presented in Fig. 5.

[8] Clang Analyzer Scan Build: https://clang-analyzer.llvm.org/scan-build.html.

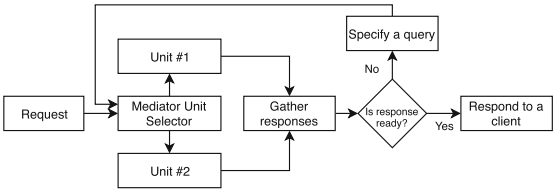

Fig. 5. Request life cycle

5 Evaluation

The proposed method was evaluated as a part of author's Equid analyzer project [7]. It is not easy to do a quantitative evaluation as differences from other models are rather qualitative, however, we are able to measure the time taken for specific parts of a run.

We conducted two experiments on the analyzer, comparing a standalone version with resident analyzer:

1. Check the full time for performing queries. For that test, the workspace had been already propagated, but not sent to the analyzer. We used a 300KLOC proprietary project and an Intel Core i7-7700HQ-based computer with 16 GB RAM, NVMe SSD and Ubuntu 18.04. The analyzer was instrumented to measure the time required. The formula for query time T is as follows:

$$T = T_w + \sum_{i=1}^{N} \cdot (T_{c_i} + T_{f_i} + T_{p_i}) + \sum_{j=1}^{M} T_{q_j}$$

where T_w, T_c, T_f, T_p, T_q are times to perform workspace creation, controlled compilation, fetching, preliminary analysis and analysis on demand, respectively. N designates a number of files in the workspace, M is a number of files required for a query to succeed. This formula doesn't take operations like dependency check, minor intercommunication periods into account because otherwise, the formula would be unreadable. For the standalone version, we assume that whole bringup is required for all tests. For the resident version, we assume that bringup is required only once, therefore this part is not included in a calculation.

   (a) Query #1: check that the return value of all functions is always initialized. Summaries can be used, but the initialization check is more consuming.
   (b) Query #2: check if there are functions with the constant return value. The summaries are used, therefore the check is cheap.
   (c) Query #3: verify that all functions calling **mutex lock** also call **mutex unlock** afterward (no matter how it is called in the target operating system). This check is flow-sensitive and requires a deeper look into Virtual Machine codes, but the analyzer preserves metadata and therefore is able to filter out functions not using mutexes at all.

Table 1. Time required for queries in case of standalone and resident analyzers

Test	Time (ms)			Difference (%)
	Standalone (1)	w/o bringup (2)	Resident (3)	(2) − (3)
Initial bringup	12536	0	12682	–
Query #1	14854	2318	2772	−19.5
Query #2	12673	137	194	−41.6
Query #3	15742	3206	3430	−6.9

Table 2. Time required for queries in failover case

Test	No failure case (ms)	Rescheduling case (ms)	
		Shared database	Local database
Query #1	2318	3709	14515
Query #2	137	210	10273
Query #3	3206	3857	13929

Success criteria: resident analyzer is faster or at most twice slower than a standalone version on the evaluation without a bringup stage.

The results are presented in Table 1. Initial bringup is more or less on par (around 1% difference), but standalone version queries are definitely more expensive than queries to a resident analyzer. For the visibility, we subtracted a bringup stage from standalone version results. In this case, resident analyzer acts 6.9–41.6% worse, but as the algorithms are the same for both cases, it is a direct result of query scheduling.

2. Check the time needed to reschedule the execution which failed in the middle of operation (around 50% of corresponding query time). "No failure" case is measured for a shared database. A few nodes of similar processing power (the same CPU as in the previous test) were used, connected by 1 gbps Ethernet using a star topology.

Success Criteria: the execution continues after the failure and is taking no more than twice a time of a regular case.

Saving to a shared database in these tests is 1% slower than to a local one, so a separate measure is not provided. "Rescheduling" is a case when *all operating nodes fail* (4 nodes are running among 8 available) and the execution is transitioned to other nodes. The overview is provided in Table 2. It is observed that rescheduled execution is taking 20%–60% of the original time depending on the non-determinable stage of the nodes. Complete rescheduling in case of local database is taking a bringup time minus supposed T_w, T_c, so a federation of separate analysis units doesn't pass the test.

6 Discussion

The results presented in a previous section hint that the model is good enough for its use cases. The best part is that a new execution mechanism handles real-time queries much better compared to a standalone scheme. Comparison-wise, it would be the same as IDE-bound server, thus, in general, the model equals the *language server* approach with a distribution.

The validity of the described method relies on static analyzer implementation. Our tool supports load distribution natively, so the method naturally extends it. The schema is not "drop-in" for all analyzers, in some architectures, the performance penalties will overweigh the benefits of having the analysis resident.

Some of the details must be finalized by the community. A common Analysis Server Protocol defining request models, query format and result representation should take more uncovered industrial applications into account. Therefore the described communication way is not a final solution, but rather an invitation to start the discussion.

7 Conclusion

We presented a model of a resident static analyzer, comprising a mediator and analysis engines. The mediator is backed by a workspace database containing build recipes and target environment details. Each analysis engine uses a local database and has access to a common database. That makes it possible to serve real-time user requests. The model was tested quantitatively and acted with a 6.9–41.6% slowdown compared to a standalone analyzer with a predefined analysis plan, which is expected as the latter is meant to be random for the resident analyzer. The failover mechanism helped avoid data loss on analyzer unit failure at the cost of additional computation time. These results are important for an analyzer project which is meant to stay at the background and respond to queries.

References

1. Language Server Protocol. https://microsoft.github.io/language-server-protocol
2. Bessey, A., et al.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (2010)
3. Campbell, G., Papapetrou, P.P.: SonarQube in action. Manning Publications Co., Shelter Island (2013)
4. Dean, J., Ghemawat, S.: MapReduce: A flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
5. Dubey, A., Wagle, D.: Delivering software as a service. McKinsey Q. **6**(2007), 2007 (2007)
6. Gasior, Ł.: ReSharper Essentials. Community Experience Distilled. Packt Publishing, Birmingham (2014)

7. Menshchikov, M.: Hybrid system of static analysis with proof-based verification of invariants. Master's thesis, Saint Petersburg State University (2018)
8. Menshchikov, M.: Scalable semantic virtual machine framework for language-agnostic static analysis. In: Distributed Computing and Grid-technologies in Science and Education, pp. 213–217 (2018)