Eilenberg Distributed Compiler Architecture

1st Emyle Santos Lima
Computer Science Department
UFMG
Belo Horizonte, Brazil
emylesantoslima@dcc.ufmg.br

2nd Fernando Magno Quintão Pereira
Computer Science Department
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br
Advisor

3rd Ian Trudel
Ian Trudel Research Center
Quebec, Canada
ian.trudel@gmail.com
ian.trudel@researchcenter.io
Advisor

***Abstract*—As software systems continue to grow in size and complexity, efficient compilation strategies have become increasingly critical. Existing approaches, such as incremental and distributed compilation, offer significant performance improvements but are typically implemented as external tools layered on top of traditional compilers. This project presents a Distributed Compiler Architecture for the Eilenberg programming language, which is inspired by Category Theory. Unlike conventional distributed build systems that merely coordinate remote compilation tasks, the proposed architecture integrates both incremental and distributed compilation directly into the compiler's core design. By parallelizing all compilation phases, including lexical, syntactic, and semantic analysis, optimization, and code generation, the architecture aims to enhance scalability, efficiency, and overall performance for large-scale applications. This work details the architecture's components, describes their interactions, and outlines the implementation plan for its primary elements.**

***Index Terms*—distributed compilation, incremental compilation, distributed compiler architecture, build systems**

I. Introduction

Over the years, advancements in technology have driven a substantial increase in the size and complexity of software systems. A notable example is Google's codebase, which in 2016 exceeded 2 billion lines of code across more than 9 million source files in 2016, encompassing approximately 1 billion files within a single repository [1]. Managing systems of this scale would be infeasible without efficient workflow strategies, especially with regard to compilation, a process executed repeatedly throughout the development cycle.

To address compilation inefficiencies, several techniques have been developed. *Incremental compilation* reduces unnecessary computation by avoiding the recompilation of unchanged portions of the program, while *distributed compilation* accelerates the process by parallelizing compilation tasks across multiple machines. Distributed build systems [2] [3] [4] typically combine these techniques by layering file distribution mechanisms on top of existing compilers.

The Eilenberg Distributed Compiler Architecture is designed around the Eilenberg programming language [5], which is inspired by Category Theory and aims to empower users to express sophisticated operations through abstract mathematical frameworks in a simpler and more efficient way. To handle the complexity of large-scale applications, the architecture integrates scalability, efficiency, and other key features directly into the compiler itself, rather than relying on external tooling, in order to deliver better performance.

Therefore, this Computer-Oriented Project focuses on the development of a *Distributed Compiler Architecture* for Eilenberg that enhances the compilation process using both incremental and distributed strategies. More specifically, we designed the architecture by describing its constituent components and their interactions, and then implemented a simple prototype including only the core features, in order to obtain a minimally testable system. However, due to time constraints, it was not possible to implement all of the intended features. Nonetheless, a plan for implementing the remaining features is presented.

Our proposed architecture differs from existing distributed build systems, which typically operate as external coordinators managing the distribution of files to remote machines, where conventional compilers perform the actual compilation. Such systems effectively function as front-ends to existing compilers. In contrast, our approach integrates distributed compilation directly into the compiler architecture itself, enabling the parallelization and distribution of all compilation phases, including lexical, syntactic, and semantic analysis, optimization, and code generation.

The remainder of this document is structured as follows. Section II reviews related work on distributed compilation and other techniques aimed at improving compilation efficiency. Section III outlines the process used to define the architectural requirements. Section IV presents the proposed distributed compiler architecture, detailing its main components and their interactions. Section V then discusses the features included in the prototype, describing both the implemented functionality and the plan for completing the remaining components. Finally, Section VI summarizes the conclusions of this work.

II. Related Works

A. Build Systems

Build systems [6] are tools that automate repetitive tasks involved in software development, particularly compilation and the overall build process. They orchestrate activities such as source code compilation, dependency management, and the generation of final artifacts, thereby promoting consistency, repeatability, and efficiency. Two fundamental design decisions underlie most build systems: determining the order in which tasks should be built, and deciding whether a given task needs to be (re)built or not. These decisions work together to provide greater efficiency in the software build process. The following subsections discuss how different build systems address these choices.

1) Make: Make [7] is a well-established build system that automates software building using *Makefiles*, which specify tasks and their dependencies. It executes each task at most once and only when its dependencies have changed since the previous build. To detect which files require rebuilding, *Make* relies on file modification times. To determine the execution order, it constructs a task dependency graph from the *Makefile* and executes them in topological order, ensuring that each task runs only after all its dependencies have been solved. Because all dependencies must be declared in advance, *Make* does not support dynamic dependencies.

2) Shake: Shake [8] is a build system designed to support dynamic dependencies. Instead of relying solely on file modification times, it records the dependency graph from the previous build to determine which tasks need to be re-built. When no prior graph is available (as in the initial build), *Shake* begins by running tasks in an arbitrary order. If, during execution, it discovers a new dependency that has not yet been built, it *suspends* the current task until the required dependency has been resolved.

B. Distributed Build Systems

Distributed build systems are tools that enable source code to be built in a distributed and parallel manner across multiple machines or servers. By dividing work among several hosts rather than relying on a single machine, they can significantly accelerate the build process, especially for large software projects. The following subsections provide an overview of various existing distributed build systems.

1) distcc: distcc [2] is a program that distributes the compilation of C-family source code across multiple machines using a client-server architecture. The client receives user commands and delegates compilation tasks that can be executed in parallel to remote servers. Each server then performs its assigned compilation task and returns the results, such as object files or error messages.

distcc distributes only the compilation phase of the build process: preprocessing is always performed locally to avoid repeatedly transferring header files over the network, thereby reducing communication overhead. After preprocessing, the source file is either compiled locally or sent to a selected host. Makefile processing and linking are always executed locally.

Task assignment is based on a lightweight load-balancing strategy: each client maintains a status file tracking active jobs and server rejections, while servers limit the number of concurrent connections to prevent overload.

distcc can also be used together with the caching system *ccache* [9], which stores compiled object files alongside their corresponding source files. This integration enables rapid detection of repeated builds and reuse of previous results, thereby avoiding unnecessary recompilation.

2) distcc pump mode: distcc with *pump mode* [10] extends the basic system by enabling the preprocessing phase to be distributed as well. Pump mode identifies the files needed for preprocessing and sends them to the server nodes. By offloading preprocessing, pump mode further reduces the client's workload and increases the overall parallelization potential, resulting in additional performance improvements.

3) Icecream: Like *distcc*, *Icecream* [3] distributes compile jobs across multiple machines to enable parallel builds. However, unlike *distcc*, Icecream employs a central scheduler that dynamically assigns compilation tasks to the fastest available server, improving resource utilization. Icecream can also be combined with *ccache* to benefit from cached build results.

4) DiSCo: DiSCo [4] is a Distributed Scalable Compiler Tool that distributes all phases of the build process, including preprocessing, compilation, and linking, across remote machines. As a result, it offers greater parallelization opportunities than *distcc*, potentially leading to even faster compilation times. Its architecture consists of a master node and multiple remote nodes. The master node contains the source code and coordinates the distribution of compilation commands to the remote nodes.

To balance the workload across nodes, *DiSCo* proposes a static scheduling strategy. The master node performs extensive dependency analysis and groups independent compilation commands into so-called *command blocks*. Rather than sending commands individually, as in *distcc*, the master assigns sets of independent commands to remote nodes, reducing scheduling overhead and improving parallel efficiency. Additionally, to further minimize communication costs, remote nodes maintain a caching mechanism that stores frequently accessed files, thereby reducing repeated data transfers over the network.

5) Bazel: Bazel [11] [6] is an open-source cloud-enabled build system that supports the use of a remote cache, allowing team members to share build results and thereby significantly accelerating the build process. The remote cache stores two types of data: (i) a *content-addressable cache* of output files, mapping each file's content hash to its corresponding artifact; and (ii) a history of executed build actions, annotated with the hashes of their input and output files. The action history enables Bazel to determine whether a matching build output already exists, while the content-addressable storage allows retrieval of the corresponding artifacts when needed.

If a required action has no matching entry in the remote cache, Bazel executes the action and produces the necessary outputs, which are then stored for future reuse. Although Bazel builds are executed locally by default, its remote execution capabilities allow build actions to be distributed across remote machines, such as a cluster or datacenter, providing additional opportunities for parallelism and improved performance.

C. Cache Systems

Many distributed build systems employ caching systems to avoid recompiling unmodified portions of a program by reusing previously cached results. These systems typically inspect textual or metadata changes in source files to construct a dependency graph that determines which tasks must be rebuilt.

However, not all modifications to the source code lead to changes in the program's behavior. For example, inserting blank lines, comments, or even unused variables, does not alter a program's semantics. Consequently, files containing only such modifications do not necessarily require recompilation. This observation motivates more sophisticated approaches that operate at the semantic rather than textual level.

1) c-Hash: c-Hash [12] addresses this limitation by detecting redundant builds through hashing the program's *Abstract Syntax Tree* (AST), rather than hashing the raw source files as done by traditional tools such as *ccache*. By computing hashes at the language level, *c-Hash* can better detect constructs that actually affect the compilation outcome.

c-Hash operates after semantic analysis and generates a hash from the resulting semantically enriched AST. The hash is computed using a depth-first traversal beginning at the top-level definitions, combining the hashes of referenced nodes with key local properties. Because semantic analysis introduces cross-tree references between related AST nodes, thereby transforming the AST into a directed graph[1], *c-Hash* incorporates mechanisms for safely handling cycles. The final hash captures the program's semantics, limited to top-level reachable elements.

2) Cloud Cache Systems: Another strategy for improving caching performance in build systems is to support cache sharing across development teams. Shared caches reduce redundant storage and enhance team productivity by enabling developers to reuse one another's build outputs. *sccache* [13] [14] exemplifies this approach: it is a compiler caching tool similar to *ccache* but capable of storing results either locally or in various cloud storage backends. Another example is *Bazel*, which allows users to configure a server as a remote cache for build outputs, enabling developers to reuse artifacts produced by team members instead of rebuilding all of them locally.

III. Architecture Requirements Definition

As mentioned in Section I, the Eilenberg Distributed Compiler Architecture (EiDCA) aims to address the complexity inherent in large-scale applications, making efficiency and scalability central objectives. To this end, several existing *distributed build systems* (see Section II) were examined to identify design principles that could guide the development of the proposed architecture.

The primary considerations in designing a distributed compilation system include determining which units must be recompiled – thus avoiding redundant compilation and improving efficiency – and establishing the correct execution order to ensure determinism and replicability. Additional factors involve identifying the information required to support these decisions and the mechanisms for obtaining or storing it (e.g., through a caching system), as well as developing strategies for distributing compilation tasks across remote nodes in a manner that minimizes scheduling and communication overhead.

It is important to emphasize that, although the systems analyzed share some similarities with the architecture proposed here, they are not compilers themselves. Tools such as *distcc* [15] function as front-end interfaces to existing compilers like gcc and clang, while *DiSCo* similarly serves as a front-end to compilers such as gcc and tcc, as reported in [4]. These systems act primarily as coordinators of the build process, distributing source files across remote nodes, while the actual compilation steps are carried out using traditional compilers. For this reason, in this project, we refer to them as *distributed build systems* instead of *distributed compilers*.

In contrast, our objective is to develop a new distributed compiler that not only distributes files across network nodes but also parallelizes all stages of the compilation pipeline, including lexical, syntactic, and semantic analysis, optimization, and code generation. In doing so, the proposed architecture integrates distributed compilation as a fundamental design feature rather than layering distribution around an existing compilation toolchain.

IV. The Distributed Compiler Architecture Design

Our proposed distributed compiler consists of a client-server architecture. The *client (eic)* is the interface used by developers to interact with the compiler. It sends all build requests to the server and awaits the compilation results. All the compilation happens in the server, offloading the build process from the client. The *server (eis)* handles all build requests from clients, managing remote compilation, optimization, and caching. It is composed by a *master node*, that distributes the compilation across several *remote nodes* for faster build of projects. The master also manages a cache system to avoid recompilation of unchanged files and categories to speed up future builds. Figure 1 shows an overview of our distributed compiler architecture with its main components. The arrows between the components represent the interactions and communication paths within the system. The following subsections provide a more detailed explanation of each component.

A. Eilenberg Client Components

1) Command Line Interface: The *Client Command Line Interface (CLI)* parses user-entered commands, provides help and command discovery, and displays log output and compilation status.

2) Client Project Manager: The *Client Project Manager* handles the storage of source files, binary files, logs, and debugging artifacts for each developer's project. These data are stored locally on the developer's machine, and the necessary files are sent to the server whenever the user initiates a compilation request.

B. Eilenberg Server Master Node Components

1) Command Line Interface: The *Server Command Line Interface (CLI)* is used by server administrators to manage

[1]Semantic analysis adds cross-tree references between related AST nodes, effectively turning the AST into a directed graph.

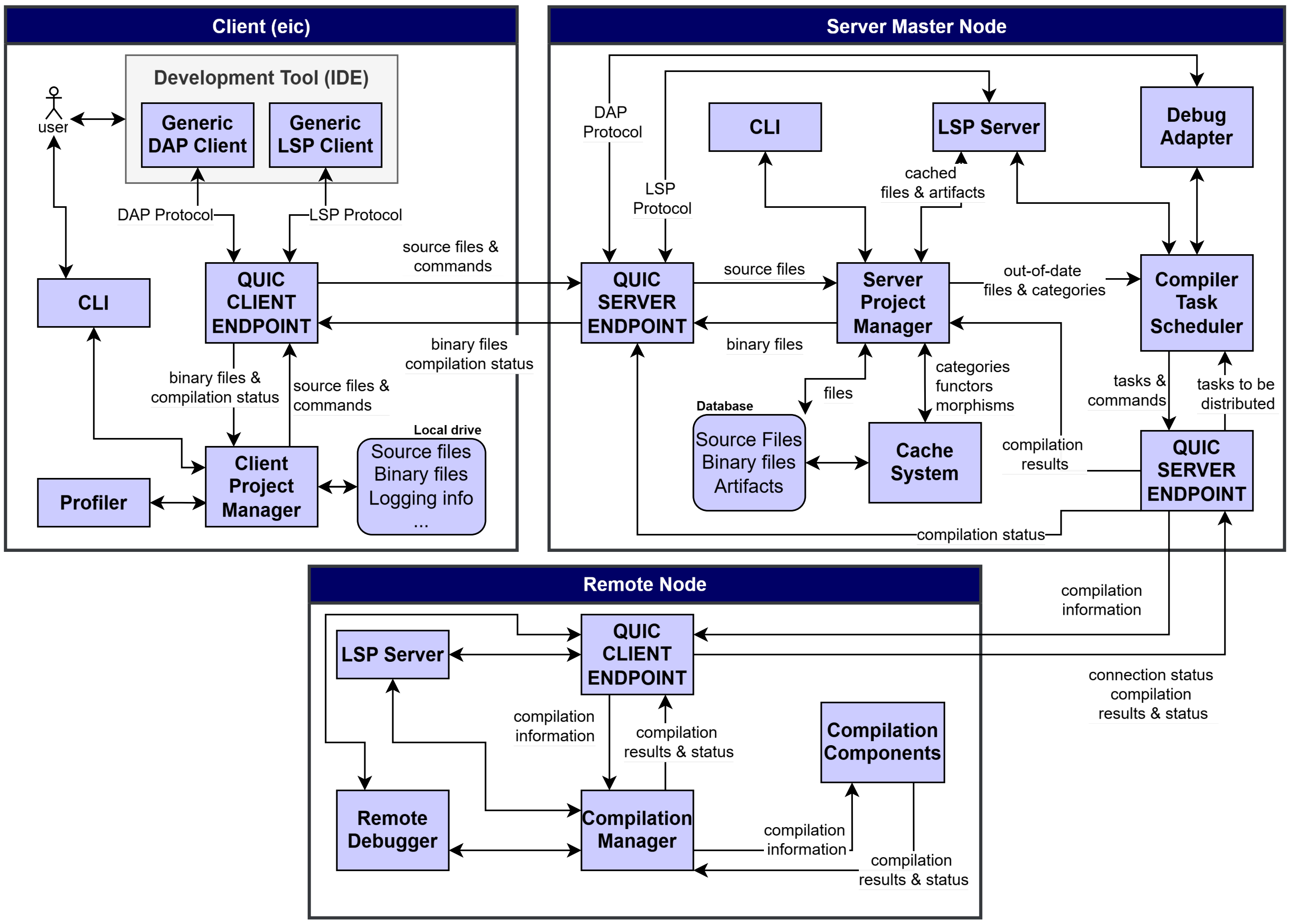

Fig. 1. The Eilenberg Distributed Compiler Architecture.

and control the server. It parses user-entered commands, supports help and command discovery, and displays logs and compilation status information.

2) Server Project Manager: The *Server Project Manager* is responsible for managing the storage of previous compilation results for each project under development to avoid the recompilation of unmodified parts of the project, such as categories, functors, or even entire files. This includes binary files and other artifacts generated during compilation, all of which are stored in a database to achieve greater scalability and robustness. When the Client requests a compilation, the system first checks for a previously stored valid result. If one is available, it returns the cached result to the Client, avoiding redundant recompilation of the same source.

3) Cache System: The *Cache System* stores categories, functors, and morphisms in memory during project compilation to accelerate access times. When a key is requested, the system first looks it up in the *Cache Hash Table*, as shown in Figure 2. This hash table indexes categories and related structures by their ID computed from the HDG (see Section IV-C1 for more details). If the key is found (a cache hit), the corresponding value is returned immediately. If it is not found (a cache miss), the system retrieves the value from the database and updates the hash table with the new key for future queries. If the key is missing from the database as well, the related structure is considered out of date and must be recompiled.

4) Database: The *Database* stores binary files and artifacts, and their related source files or categories and functors. It maintains both a *stable version* of each project, accessible to all users working on that project, and *user-specific versions*, which are accessible only to the corresponding user. Files are indexed based on their *Content Identifiers (CIDs)* [16]. This mechanism enables efficient detection of changes: any modification to a file results in a different CID. When a search is performed, the system first searches the user's personal version. If the key is not found there, the search continues in the project's stable version. When a successful compilation of the project is completed, the user changes are incorporated into the stable version of the project.

5) Compiler Task Scheduler: The *Compiler Task Scheduler* is responsible for scheduling compilation-related tasks to be distributed across remote nodes. Primarily, it determines which keys need to be recompiled and in what order. A key should be recompiled only if it is out-of-date, which means it is either not stored in the server or it has a dependency on another out-of-date key.

The Scheduler operates by requesting out-of-date keys from the *Server Project Manager* and distributing them to be compiled on remote nodes. Additionally, the Scheduler may also receive tasks that need to be rescheduled from the remote nodes in two situations. The first is when a remote node fails and cannot return its compilation results, in which case its tasks must be redistributed to other nodes. The second situation occurs when a task that is already running on a remote node can be parallelized and redistributed again to more nodes to speed up the build time. In this case, it is the remote node that requests the redistribution.

The Compiler Task Scheduler is essential to ensure that only modified categories and functors are recompiled, thereby avoiding unnecessary work and speeding up the build process. However, since it is not possible to determine in advance which categories have been modified, the following strategy is adopted: entire source files are processed until the syntactic analysis is completed, at which point it becomes possible to identify the categories and functors contained within the file. The system then checks which of them are out-of-date, and only those proceed through the remaining compilation phases. To make this possible, the Compiler Task Scheduler interacts with the Project Manager at two distinct points during the compilation process. First, it requests out-of-date files and distributes them to remote nodes to undergo lexical and syntactic analysis. During the last phase, a *Hybrid Directed Graph (HDG)* is generated (see Section IV-C1 for more details), which allows the identification of all categories and functors contained in the file. Once this information is available, the Compiler Task Scheduler queries the Project Manager to determine which of these are out-of-date and redistributes them to complete the remaining compilation stages.

Nevertheless, keys that depend on out-of-date ones also need to be recompiled. Since it is not possible, in principle, to generate a complete dependency graph in advance, as *Make* does (see Section II-A), to identify the dependent keys, the Compiler Task Scheduler, together with the Compilation Manager, uses the HDG to check the dependencies between categories. Any category requiring recompilation is forwarded to the Scheduler and then distributed accordingly.

Finally, if a dependency is detected during task execution on a remote node, two strategies can be used: (i) stop the task and restart it once the required dependency has been resolved; or (ii) suspend the task until the dependency becomes available. The choice between these strategies depends on the trade-off between the cost of suspending a task versus restarting it (potentially multiple times), and will require empirical evaluation in the future.

The Compiler Task Scheduler is also responsible for scheduling tasks related to the execution of the *Language Server Protocol* (LSP) (see Section IV-E2) and the *Debugger* (see Section IV-E3) on remote nodes. However, a detailed explanation of these mechanisms is beyond the scope of this work at this time.

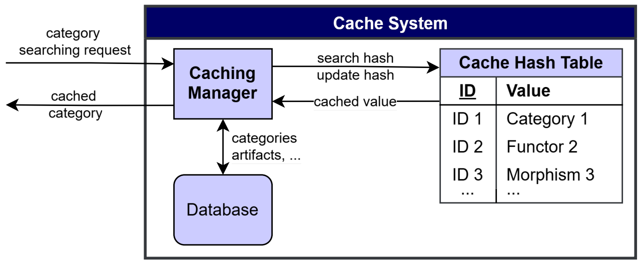

Fig. 2. The Server Cache System.

C. Eilenberg Server Remote Nodes Components

1) Compilation Manager: The *Compilation Manager* is responsible for managing the entire compilation process. It receives compilation tasks from the server master node, delegates them to the appropriate compilation components, monitors their progress, and redistributes tasks when necessary.

Figure 3 illustrates how the Compilation Manager interacts with the various compilation components. The most notable difference between this pipeline and that of traditional compilers is that Eilenberg uses a *Hybrid Directed Graph* (HDG) as its intermediate representation (IR), instead of relying on an *Abstract Syntax Tree* (AST) as most programming languages do.

The HDG combines both control flow and data flow into a unified structure, enabling more sophisticated analyses, such as the identification of independent compilation units, i.e., portions of the program that can be processed in parallel. For instance, if a graph node depends only on nodes A and B, and both A and B have already been processed, this node can be sent to the server master node to be compiled on a remote node.

The compilation pipeline is organized into several interconnected stages, each of which can be executed independently and in parallel. It begins with the lexical analyzer, which takes a source file as input and produces a stream of tokens. These tokens are then consumed by the parser, which constructs a HDG. The semantic analyzer operates directly on the HDG, performing checks such as type validation, symbol resolution, and other semantic analyses. Subsequently, the optimizer applies transformations to improve performance, producing an optimized HDG. Finally, the code generator receives the (optimized) HDG and emits binary files targeting the desired architecture.

It is important to note that the pipeline is designed to enable parallel execution in all stages. For example, while one source file is undergoing lexical analysis, another might already be in the optimization phase. Similarly, multiple independent HDG components, like independent categories or functors, can be optimized or code-generated concurrently. This architecture enables efficient workload distribution across multiple remote nodes, supporting scalable and high-performance compilation.

Whenever a stage finishes execution, it reports its status (completed or failed) along with any resulting data to the compilation manager. The manager then initiates the next stage and

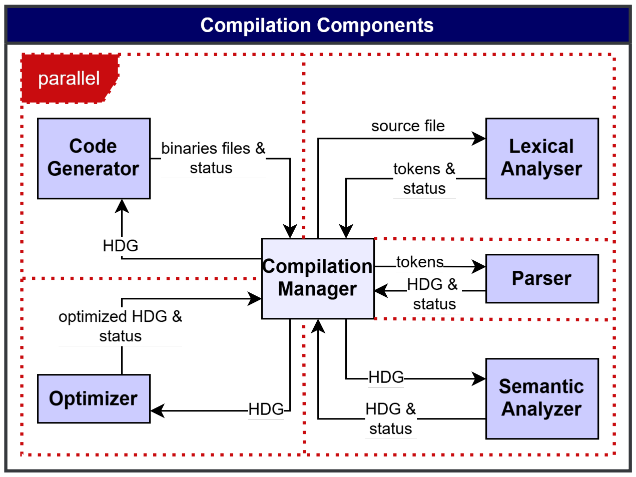

Fig. 3. Interaction between the Compilation Manager and the compilation components. The arrows indicate the information exchanged between them. The Parallel frame represent that each component can be executed concurrently.

may choose to distribute its execution across additional remote nodes, based on current workload and system conditions.

Additionally, the compilation manager has the authority to interrupt or terminate the execution of any ongoing stage under specific situations. These include, for example, the discovery of an uncompiled dependency that must be resolved before proceeding; a compilation failure on one or more nodes that prevents continuation or recovery, prompting an early termination to avoid wasting computational resources; or a strategic decision to redistribute the current task across other nodes to accelerate the overall compilation process.

D. Communication Client-Server

Communication between the client and server will use *QUIC* [17] with TLS 1.3 support. QUIC is a transport protocol developed by Google to enhance HTTPS performance through faster handshakes, mandatory use of secure ciphers, and the elimination of the head-of-line blocking issue present in HTTP/2. It combines transport layer functions, TLS 1.3 encryption, and certain application-layer features. Like TCP, QUIC is connection-oriented, reliable, and includes built-in flow and congestion control.

We chose QUIC for its performance and security advantages, as large volumes of data will need to be transmitted over the network during builds and compilations. Our QUIC server-client setup will include two kind of channels: one for basic communication, primarily for sending commands and status updates, and another for data transfer, dedicated to transmitting files. This separation is essential because monitoring the compilation and connection status requires fast, responsive communication. Mixing it with heavy data streams, such as large files, could introduce latency and delay the delivery of critical commands and status messages.

E. Tooling

The compiler will provide tools that help developers write and debug their projects more efficiently. Due to time constraints, these tools will not be developed as part of this project. However, we have dedicated a section to present the main idea behind each of them.

1) Profiler: The *Profiler* is a tool designed to measure project performance, identify execution bottlenecks, and suggest improvements to enhance efficiency.

2) Language Server Protocol: A *Language Server* is responsible for providing language-specific intelligent features such as code auto complete, go to definition, and syntax highlighting, helping developers write code more efficiently. The *Language Server Protocol* (LSP) [18] is the protocol that standardizes communication between a Language Server and a *Language Client*, which is implemented within a development tool. This standardization allows a single Language Server to communicate with different development tools, and a single Language Client to interact with Language Servers for different programming languages. The Language Client is unaware of the details of any specific language and therefore offers only language-independent features, such as presenting files, navigating through lines, and adding or deleting characters. Features that depend on language-level knowledge are provided by the Server, since it needs to interact with the AST or other intermediate representations (IR) to determine, for example, whether a given element is a class or a function [19].

In the case of Eilenberg, the Language Server will need to interact with the HDG (See Section IV-C1), which is the IR used, and therefore with the compilation components on the remote nodes. As a result, the LSP Server execution will be distributed among remote nodes, which can increase efficiency by enabling parallel execution. This LSP Server aims to provide developers with smarter code auto complete suggestions that help improve project performance by recommending more efficient morphisms.

3) Debugger: A *Debugger* is a program utilized for debugging other programs, helping developers analyze the running code to identify and fix bugs. The most common debugging technique involves setting breakpoints and suspending the program's execution when a breakpoint is reached. At this point, the developer can inspect the state of the program, such as its variables, and run it step by step. This method is known as *Trace Debugging* [20], and it is a more manual process. Many other techniques have been developed over the years with the aim of making the debugging task faster and more precise, but discussing them is beyond the scope of this work.

Originally, each programming language had to implement a different debugger to communicate with the user interfaces of various development tools. To address this issue, Microsoft developed the *Debug Adapter Protocol* (DAP) [21], a protocol designed to standardize this communication, eliminating the need to implement tool-specific debuggers. The core idea is that each development tool includes a generic debugger that communicates with a debug adapter using the DAP to exchange the necessary information for the actual debugger to perform its tasks. The debug adapter acts as an intermediary component that bridges the concrete debugger with the standardized protocol.

In our distributed compiler architecture, the concrete debugger operates on remote nodes to interact with the compilation components and execute the target program in debug mode. The master node hosts the debug adapter, which serves as a bridge between the concrete debugger and the IDE's generic debugger on the Client side. This design not only enables seamless integration but also enhances efficiency, as debugging tasks can be distributed across remote nodes and executed in parallel.

V. Implementation

The goal of this part of the project was to implement the core features of the EiDCA, producing a minimally functional system suitable for testing. Although full implementation of the planned scope was not achieved, the subsections below describe the technologies used, the features that were initially intended for development, the components that were successfully implemented, and the planned approach for those that were not.

A. Tools and Technologies

Originally, the architecture was intended to be implemented using the Eilenberg language itself; however, because its final version is not yet complete, we opted to use *Rust* for this phase of development. Each unit of the architecture (client, server master node, and server remote node) has its own dedicated subproject.

B. Command Line Interface (CLI)

The implementation of the CLI centered on providing commands to manage the EiDCA features developed in this project, such as sending compilation requests on the client side and performing basic server operations (e.g., starting the server master and remote nodes). The CLI was built using the *clap* crate, the most widely used library in the Rust ecosystem for parsing and validating command-line arguments and subcommands. It also automatically generates documentation for help commands.

On the client side, a configuration file is used to simplify command usage by storing project properties, such as name, ID, and compilation settings. To request a compilation, the user must specify the directory containing this configuration file; if no directory is provided, the program searches for it in the current working directory.

On the server side, commands were created solely to start the master node and a remote node. For simplicity, the remote server node has its own CLI as well, rather than being controlled through the master node's CLI.

C. Basic communication

The goal was to implement basic communication using QUIC to support the execution of compilation requests. The chosen crate was *quinn*, a pure-Rust, async-compatible implementation.

The *eic* client contains a client endpoint that connects to the master server node. The remote node is also composed of a client endpoint, which simplifies connection management with the master node. Consequently, the master node exposes two server endpoints: one dedicated to handling connections from the *eic* client, and another dedicated to handling connections from remote nodes, each bound to a different port. This approach provides better isolation and message handling, since the connections do not compete with each other and each has a well-defined processing path. Furthermore, modeling remote nodes as clients rather than servers simplifies certificate management (since all connections must be encrypted and servers need to generate certificates). With this design, one or two certificates are sufficient for the two server endpoints on the master node.

In our implementation, clients always establish secure connections using the server-generated certificate, although this is not mandatory in Quinn. The *eic* client connects to the server only to execute a command (e.g., a compilation request) and disconnects once processing ends. Remote nodes, on the other hand, connect to the master server when they start up in order to register, and then remain in a loop waiting for tasks to execute.

QUIC is highly flexible, allowing us to define our own message-exchange protocol. As described in the design section, our communication requires two distinct channels to separate commands from file transfers. These channels were implemented as follows:

1) Commands: Commands are sent over bidirectional streams, since a response is usually expected (for example, sending a compilation request requires receiving the compilation result). Messages consist of a JSON object containing the command data, prefixed by its length in bytes to allow correct deserialization:

$$\text{Message} = \text{len(bytes)} \,||\, \text{JSON(command)}.$$

2) Files: Files are sent over unidirectional streams, as no response is expected; QUIC automatically handles ACKs in the background. For each file sent, we first transmit a header containing the file metadata such as filename, size, and a content hash used as a checksum. This hash not only ensures that the file was transmitted correctly, but is also used by the project manager to search for previous compilations results that may already exist. The header is encoded as JSON and, as before, prefixed with its length:

$$\text{Message} = \text{len(bytes)} \,||\, \text{JSON(file metadata)}.$$

After the header, the raw file bytes are written to the stream buffer. Because streams are cheap to create and disposable in *quinn*, and because QUIC provides multiplexing without the head-of-line blocking issue, a separate stream is created for each file, allowing them to be processed in parallel without interfering with one another.

D. Project Manager

The *Project Manager* is responsible for storing and retrieving each project's data on both the client and server sides, as well as handling the basic pipeline of the compilation process. The initial implementation plan for this component consisted of the following stages, of which only the database stage remains incomplete:

- Server-Side (*eis*):
  - Implement a UUID-based project identification,
  - Implement file receiving and storage system,
  - Create a database where files are indexed by their content,
  - Create artifact storage and retrieval system.
- Client-Side (*eic*):
  - Implement project submission with auto-generated UUIDs,
  - Create local file gathering and packaging,
  - Add checksum verification for uploads,
  - Implement compilation request initiation.

To submit a compilation request, the client first retrieves the project's UUID from its corresponding configuration file. If no UUID is found, a new one is generated randomly and saved. Then, all files targeted for compilation are collected from the project directory in the local file system. Only their metadata, such as name, size, and checksum hash, is sent to the server as part of the compilation request. The hash is computed by applying the SHA-256 digest function to the file's raw byte contents. This allows the server to check which files it already has cached and request only the missing ones, preventing unnecessary file transfers over the network.

The compilation request message therefore contains the command type, the project ID, and the metadata of the input files, and is sent according to the protocol described in the previous subsection. Upon receiving this request, the server checks which files it doesn't have yet by searching for each checksum hash in the database. If it finds a perfect compilation match (i.e., exactly the same set of files being compiled together), the entire compilation process can be skipped, and the results of that previous compilation are returned to the client. Since the database is not fully implemented yet, the parts that would rely on it are currently simulated. If any required file is missing on the server, it requests the file from the client. After receiving the missing files, it proceeds to the compilation stage, which is also currently simulated. Figure 4 illustrates this workflow.

When the server needs to return the compilation results, it first sends a message to the client indicating whether the process succeeded or failed. If the compilation was successful, the resulting binary files are then transmitted as described in the previous subsection.

Although the database has not yet been implemented, the planned design is SQL-based and would include at least the following structures: a table mapping each file's hash to its content and associated metadata; a table mapping the hashes of input files (source files) to the hashes of the resulting binary files, indexed by the compilation ID that produced them; and a table mapping a file (identified by its hash) to its semantic content (categories and functors), also indexed by their corresponding IDs. The IDs of categories and functors will be computed from the HDG generated during the corresponding compilation, following an approach similar to that used for the AST described in Section II-C1. It has not yet been decided whether the actual HDG content will be stored directly in the database or kept in the server's local file system.

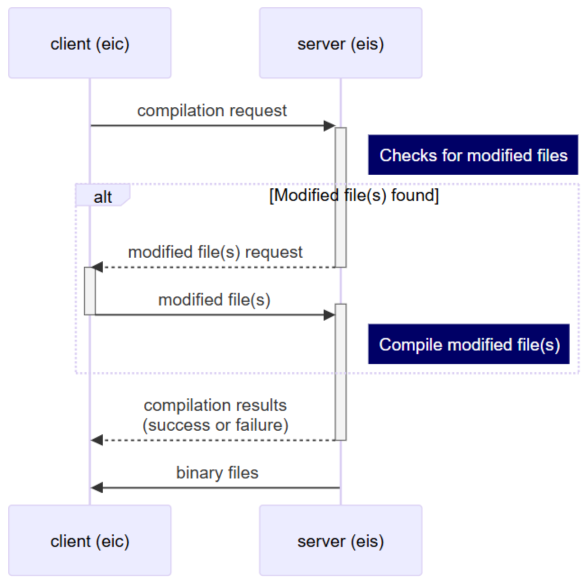

Fig. 4. Client-server workflow for compilation request. Solid lines indicates a request or a message that is sent, and dashed lines indicates a response.

For simplicity, this prototype does not implement user identification or authentication. As a result, the proposed database does not support the stable and per-user versioning scheme described in Section IV-B4.

E. Distributed Compilation Framework

The Distributed Compilation Framework is the core component of the distributed compiler. It encompasses both the *Compilation Task Scheduler*, responsible for scheduling and distributing compilation tasks across nodes, and the *Compilation Manager*, which executes those tasks on remote nodes. Its implementation involved the following steps:

- Server-Side (master node):
  - Implement a compilation task scheduler,
    - ∗ First for only one project being built at a time,
    - ∗ Then for multiple projects.
  - Implement work distribution system across nodes,
  - Implement failure recovery and task redistribution,
  - Implement node health monitoring.
- Server-Side (remote nodes):
  - Implement basic compilation pipeline,
  - Implement parallel compilation tasks execution,
  - Implement compilation status monitoring,
  - Implement compilation task interruption command.

- Client-Side (*eic*)
  - Implement distributed compilation status monitoring.

Although none of the features described above were fully implemented, we outline here the plan that would have been followed to complete them.

The *Scheduler* dynamically assigns compilation tasks at runtime to ensure a well-balanced distribution of work. It maintains a list of pending tasks and a circular list of remote nodes waiting for assignments. Each task consists of the input file, the expected output, and the command to be executed (e.g., lexical analysis, parsing, etc.). When the scheduler receives the modified files that need to be recompiled from the *Project Manager*, it generates tasks from them. The task list is ordered so that files with fewer or no dependencies are scheduled first, assuming the dependency graph is known from the previous compilation.

The list of remote nodes contains each node's ID and its available processing capacity. The scheduler iterates over this list in a circular fashion, assigning tasks according to each node's capacity and updating a mapping from tasks to nodes. This strategy assumes that only a single project is being compiled at a time. Supporting multiple concurrent projects would require modifications or an alternative strategy to avoid starvation and ensure fair distribution of resources.

When a remote node successfully completes a task, it reports the result to the master node and sends the generated output. At that point, the scheduler also updates the node's available capacity. If an error occurs during task execution, two scenarios are possible:

- If the error is unrecoverable and prevents the compilation from proceeding (for example, an invalid symbol during lexical analysis or an invalid rule during parsing), then the compilation must be aborted.
- If the error is caused by an unresolved dependency, such as during semantic analysis, when dependent files or categories may be processed on different nodes or in an incompatible order, the task is returned to the master node to be rescheduled once the dependency is resolved.

Upon receiving tasks from remote nodes (either failed tasks or new tasks generated during processing), the master node appends them to the end of the pending task queue. Tasks with unresolved dependencies can be rescheduled a limited number of times, until it becomes evident that the underlying cause is a genuine error in the code that makes the build impossible to complete.

The master node also keeps a small number of idle nodes reserved for fault recovery: if an active node fails, its assigned tasks are handed off to one of these backup nodes.

To monitor node health, the master node periodically sends ping messages and waits for corresponding pong responses.

Upon receiving an assigned task, the *Compilation Manager* on the remote node invokes the appropriate compilation component, passing along the input information received.

It is important to emphasize that not all compilation components are fully implemented (and their implementation lies outside the scope of this Computer-Oriented Project). As a consequence, the distributed compilation process would still be partially simulated to emulate the behavior of the missing components.

Compilation progress would be monitored based on task-completion messages sent from the remote nodes to the master node, allowing the system to track the overall state of the build. Status updates would be written to a unidirectional stream maintained between the server and the client throughout the compilation process.

F. Cache system

The *Cache System* stores categories, functors, and morphisms indexed by their IDs during compilation in order to accelerate access times. These IDs are computed from the HDG. Whenever a key is requested, the system first looks it up in the hash table; if it is not found there, it then checks the database or the file system, depending on where the HDG is stored. An important consideration is that the cache system does not store full files or artifacts themselves, reducing both memory usage and overhead. The implementation of the caching system remains incomplete and would comprise the following steps: implementation of a morphism-based hash table and time-to-live (TTL) and eviction policies.

VI. Conclusion

The research conducted on distributed build systems, combined with the distributed architecture proposed in this project, lays the groundwork for the full development of the architecture, which will be implemented in Eilenberg at the final stage of the research effort. By clearly identifying the essential components and defining the workflow that orchestrates their interactions, this work establishes a robust foundation for subsequent investigations. These contributions not only guide the remaining phases of the project but also provide a structured basis for advancing the study and implementation of distributed compilation systems more broadly.

References

[1] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[2] M. Pool, "distcc, a fast free distributed compiler," 2003.

[3] icecc/icecream. (2025) icecc/icecream: Distributed compiler with a central scheduler to share build load. [Online]. Available: https://github.com/icecc/icecream

[4] K. JO, S. W. KIM, and J.-K. KIM, "Disco: Distributed scalable compilation tool for heavy compilation workload," *IEICE Transactions on Information and Systems*, vol. E96.D, no. 3, pp. 589–600, 2013.

[5] I. Trudel, "Eilenberg: A morphism-oriented programming language," 2024, unpublished manuscript.

[6] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems a la carte: theory and practice," *Journal of Functional Programming*, vol. 30, no. E11, April 2020, https://doi.org/10.1017/S0956796820000088. [Online]. Available: https://www.microsoft.com/en-us/research/publication/build-systems-a-la-carte/

[7] I. Free Software Foundation. Gnu make. [Online]. Available: https://www.gnu.org/software/make/

[8] N. Mitchell, "Shake before building: replacing make with haskell," *SIGPLAN Not.*, vol. 47, no. 9, p. 55–66, Sep. 2012. [Online]. Available: https://doi.org/10.1145/2398856.2364538

[9] J. Rosdahl and A. Tridgell. (2025) ccache — a fast c/c++ compiler cache. [Online]. Available: https://ccache.dev/

[10] N. Klarlund. distcc's pump mode: A new design for distributed c/c++ compilation. [Online]. Available: https://opensource.googleblog.com/2008/08/distccs-pump-mode-new-design-for.html

[11] Bazel. Build basics. [Online]. Available: https://bazel.build/basics

[12] C. Dietrich, V. Rothberg, L. Füracker, A. Ziegler, and D. Lohmann, "cHash: Detection of redundant compilations via AST hashing," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 527–538. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich

[13] Mozilla. sccache - shared compilation cache. [Online]. Available: https://github.com/mozilla/sccache

[14] R. Matev, "Fast distributed compilation and testing of large c++ projects," in *EPJ Web of Conferences*, vol. 245. EDP Sciences, 2020, p. 05001.

[15] M. Pool. distcc: a fast, free distributed c/c++ compiler. [Online]. Available: https://www.distcc.org/

[16] F. Wang and Y. Wu, "Keyword search technology in content addressable storage system," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 728–735.

[17] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, "Quic on the highway: Evaluating performance on high-rate links," in *2023 IFIP Networking Conference (IFIP Networking)*, 2023, pp. 1–9.

[18] Microsoft. What is the language server protocol? [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/overview

[19] D. Bork and P. Langer, "Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools," *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, vol. 18, pp. 9–1, 2023.

[20] D. Ghosh and J. Singh, "A systematic review on program debugging techniques," in *Smart Computing Paradigms: New Progresses and Challenges*, A. Elçi, P. K. Sa, C. N. Modi, G. Olague, M. N. Sahoo, and S. Bakshi, Eds. Singapore: Springer Singapore, 2020, pp. 193–199.

[21] Microsoft. What is the debug adapter protocol? [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/overview