Risk Assessment and Execution Plan

Altenative Polkadot Host Node

Authors:

Ida Tucker	Zondax AG
Francesco Dainese	Zondax AG
Natanael Mojica	Zondax AG
Juan Leni	Zondax AG
Ainhoa Aldave	Zondax AG

ZONDAX AG, MARCH 2023

Contents

1 Introduction

This document evaluates the potential risks and implications of a project to create an alternate host node for the Polkadot network, using a hybrid and progressive approach. It has been written for the Web3 Foundation, since accurately assessing the risks and effort required to build an alternative host node is essential in order to allocate resources needed to successfully conduct the project.

As in any project, an inaccurate estimation of the effort involved may impact the quality of deliverables, or even result in the project being prematurely terminated. Therefore, this initial pre-engineering phase will focus on analyzing the current situation and help prepare an execution plan that takes into consideration possible risks and blockers.

A Polkadot host provides the environment in which the runtime executes, and the Polkadot network (and other related chains) have been successfully running for a few years. Though there are official specifications for the host node, there is a collective evolution and experience beyond the official specifications. We believe that this variance – between implementation and specification – explains in part why other alternative implementations of the host node have not yet reached production level quality. The existence of such a discrepancy justifies our chosen methodology: we propose a hybrid and progressive approach that involves starting with the existing Parity host node (implemented in Rust) and gradually replacing substantial areas with new C++ or Rust implementations. Such a modular approach presents less risk than starting from scratch on a new host node implementation, and will allow us to fully comprehend the role of each replaced component as we go along.

Once a component has been replaced, keeping our C++/Rust implementation up to date with the potential evolution of Parity's node implementation will be essential in order to reach production level. This will require setting up automated tools informing us of such changes. Once these tools are running smoothly, the effort of updating node components which have already been implemented should be minor relative to the overall project.

Throughout this report we present challenges identified in the proposed project. Some are independent of our chosen approach, notably the fact that the official specifications are not reliable as a unique source of information for building a production-level host node (*cf.* Sections 3.1, 3.2). This goes in hand with the fact that Parity's host node (the only host node being used at production level) evolves and may be ahead of the official specifications. We suggest (see Section 4.6) that the team in charge of the project presented in this report should actively support the specification team in bringing the specifications up to scratch.

Other challenges are directly related to our approach, in particular, as we will be starting from a fork of Parity's host node, the functionality of our code-base must stay synced with theirs. To make up for the high code churn observed in their implementation (*cf.* Sections 3.3), we suggest, in Section 4.5, means to ensure that we keep up to date. Furthermore, our choice of the C++ programming language (see Section 4.1), and of project structure comes at a cost. Implementing the interface between the fork of the parity host node in Rust, and the modules we re-write in C++, requires some thoughtful planning to avoid issues due to the high number of dependencies involved (see Sections 4.2 and 4.4). Additionally, as C++ does not have native built-in concurrency support, asynchronous calls between Rust and C++ code will require some additional development. In Section 4.3 we present a proof of concept developed to illustrate the feasibility of calling asynchronous C++ code from Rust.

For all of the foreseen challenges, we propose measures to mitigate implied risks, such as the use of specific

tooling and libraries. The resulting feeling among our team is one of enthusiasm for taking on this atypical and formidable project.

2 Methodology

In order to assess the risks associated with the project, in Section 3.1, we first inspect alternate host node implementations, to get some intuition for why most have not yet reached production level.

We then delve into the Web3 Foundation's specifications and estimate their reliability, completeness, and clarity; we further compare them to the nodes' practical implementation in `paritytech/Polkadot`.

Next, we analyze the only current production level Polkadot host node (by Parity) by splitting the code into modular components and measuring code churn and dependencies. We evaluate the stability of said components by further analyzing the impact of changes made in the `paritytech/Polkadot` repository. This allows us to identify the different components of the node to be implemented, and provide a preferential order for re-implementation.

We also detail foreseen challenges related to our choice of implementation language (C++), and notably on interaction of new crates in C++ with the original `paritytech/Polkadot` host node in Rust.

Finally, we consulted members of the Web3 Foundation, so as to get a second opinion on the major challenges involved in this project.

This work has allowed us to gain a better appreciation of the resources that will be required to succeed.

3 Assessment of existing resources

In this section, we identify challenges associated with the project. These may be internal and hence under our control (e.g. relative to the choice of programming language, missing skills...), or external and outside our control (unreliable specification, evolution of the `paritytech/Polkadot` codebase...).

3.1 Existing implementations

Production level host node. Parity Technologies has developed an implementation of the Polkadot host node, which is the central component of the Polkadot network, responsible for managing and facilitating communication between different blockchain networks. This host node (`paritytech/Polkadot`) is written in Rust and is designed to be fast, scalable, and secure. It is used in production as the reference implementation of the host node and is the most widely used implementation. It is an important part of the Polkadot network and is constantly being updated and improved by Parity Technologies and a wide community of developers. Notably, this is the *only* Polkadot host node which has reached production level.

Other host node implementations (not yet production level). Gossamer is a project developed and maintained by Chainsafe Systems, written in the Go programming language which includes its own implementation of the host node. In parallel, KAGOME, which is developed and maintained by Soramitsu, also has its own C++ implementation of the host node. Neither of these implementations have yet reached production level, and both are being written from scratch, based on the official specifications for the Polkadot host node.

As we shall see in Subsections 3.2 and 3.3, the specifications alone are not reliable enough to build a production-level host node, and the `paritytech/Polkadot` code base changes at too fast a rate (ahead of the specifications) for projects building a host node 'from scratch' to successfully adapt and follow Parity's rhythm. We believe this is one of the main reasons Kagome and Gossamer have not yet reached production level.

3.2 Review of specifications and conformance tests

We here look into the Web3 Foundation's specifications and the associated specification tests, and compare them to the knowledge/evolution of the Parity node, which goes beyond these specifications.

Stability of the specifications. The changes to the Polkadot host specification are fairly regular, and new releases occur approximately every 3 months. For example, on December 19th 2022, an updated version of the specifications was deployed (Version 0.2.1). The changes are significant, and include a reorganization of parts and chapters, adding node taxonomy and extended light client specifications, as well as removing various deprecated sections. Though updating specifications is crucial to keeping them in-line with the implementation of the only production level node, such large structural overhauls do not help in one's general understanding of the documentation.

Completeness of the specifications. Various algorithms/definitions/entire sections remain empty. It is not clear – from the specifications – how these undefined aspects of the host node should be implemented. Examples are provided in Appendix A.1.1.

Besides these undefined aspects, one may note some discrepancies in definitions (for example in the definition of an ancestor, the inequality symbol should be inverted), as well definitions that may require more clarification. When these are general definitions, recalled for ease of reading (such as that of an ancestor in a blockchain), the attentive reader can – without much difficulty – recover the correct definition. However, for definitions which are specific to the Polkadot host node, it is crucial that all terms used in a definition be well defined. Examples are provided in Appendix A.1.2.

Conformance tests. The Web3 Foundation provides specification tests for the different components of Polkadot, which may be ran against the different implementations of Polkadot. A project implementing Polkadot can thus tangibly demonstrate its growth by passing increasingly more tests in that suite.

However, many parts of the Polkadot specification for the host node are not yet covered by tests in Polkadot-tests. A few examples of untested requirements are: consensus failure (this is work in progress), discovery mechanisms (finding peers within the network), block announcements and requests, warp sync protocols, equivocation[1], and many more. Furthermore, we could not find tests ensuring that different Polkadot implementations can correctly communicate with each other.

For more details on what *is* covered in the Polkadot-tests, see Appendix A.1.3.

Loss of expertise. A lot of the knowledge is owned by a subset of individual. If and when such key figures leave the ecosystem, knowledge could be temporarily lost, with no means to quickly transfer skills and knowledge.

[1] The Polkadot Host must detect if a voter broadcasts two or more valid votes to blocks during one voting sub-round, and submit it to the Runtime

Trailblazing `paritytech/Polkadot` code-base. Changes to the host node tend to be implemented in `paritytech/Polkadot` *before* they are documented in the Web3 Foundation's Polkadot-spec, and maintainers of the specifications are not systematically informed of these changes. For example, the collation generation subsystem was first sketched out in `paritytech/Polkadot` in August 2020, whereas collations were only defined in the Polkadot-spec in July 2022. Similarly, support for warp sync was added to the `paritytech/Polkadot` host node in January 2021, whereas a definition of warp syncs was only added to the Polkadot-spec in July 2022.

Divergence between outlined protocols and implemented protocols. There is a list of protocols in the Polkadot specifications, along with a description of how the protocol specifiers are constructed. Though the protocol specifiers in the implementation fulfill the specifications regarding how they are structured, there are a number of protocols listed in the specifications that are not part of the current node implementation and vice-versa. A list of such protocols is provided in Appendix A.2.

3.3 Review of Polkadot Host (Parity)

Due to the fact that the specifications alone may not sufficient to build a host node, we will also use the code-base for Parity's host node as a reference. Indeed, as mentioned in the introduction, our approach to building a host node on the Polkadot network will be modular, starting from a fork of `paritytech/Polkadot`, and replacing components with our own C++ implementations one at a time. Our goal being to divide the task into smaller, more manageable modules, that can be implemented and tested separately. As Parity's note is written in Rust, it is cut into crates. Hence the minimal units we can replace will be these crates, and we are somewhat restricted to follow the architecture of their project.

In this section we expose challenges due to our dependence on the `paritytech/Polkadot` code base.

Code churn & Substrate dependencies. The ecosystem is thriving and `paritytech/Polkadot` code base continuously evolves (some areas more than others). This may complicate the re-writing process of modules, and implies the need for a strategy ensuring that modules, which *have been replaced*, stay up to date with Parity's host node.

As Parity's node implementation is written in Rust, crates split the code into reusable and composable pieces. We will take the crates as areas of the Polkadot host node which we believe can be implemented independently.

For evaluation of code stability, we use a tool called Hercules to get a variety of stats on the history of project files. We thus obtained an objective evaluation of which crates are most stable, as opposed to those that are subject to high levels of change. Since the project also heavily depends on external crates – namely from Substrate – we also evaluated the number of updates the `Cargo.toml` files, and the complexity and code churn of these dependencies. Details are provided in Appendix B.

The large graph of dependencies, involving crates from Substrate, presents an important risk factor to our project. Firstly because Rust's Cargo provides a more streamlined and integrated experience for managing dependencies compared to the use of third-party tools in C++; secondly because some of the Substrate crates will also require re-implementing; and finally, as we will be moving some parts of the host node's code to C++, while still having crates in Rust which depend on the original (Rust) crate, these dependencies will most likely cause a number of compile-time errors (e.g. in the event of changes on dependencies or types).

Rust macros. Parity's Polkadot node makes extensive use of macros in Rust throughout the code, which will make the task of re-writing the project in C++ more challenging. This is because macros in Rust are expanded at compile-time, and they can generate complex code that may not be straightforward to translate to C++.

To re-implement macros in C++, we will need to manually translate the macro logic to C++ code. This can be a time-consuming and error-prone process, as we will need to understand the behavior of the macros and ensure that the C++ code behaves the same way.

4 Proposed approach

In this section, we discuss the strategies that we will be using to manage the identified risks. In particular, in Subsection 4.1, we first justify our choice of C++ as programming language to re-write the node. Then in Subsection 4.2, we detail the interface that we have setup, allowing our fork of the Parity node (in Rust) to depend on crates which will be re-written in C++. Next, in Subsections 4.3 and 4.4 we discuss our approach to handle the risks related to asynchrony and dependencies. Finally, in Subsections 4.5 and 4.6 we explain how we intend to keep up to date with Parity's implementation of the Polkadot host, and how to mitigate the risks related to unreliable documentation.

4.1 From Rust to C++

Given that the original node is written in Rust, to write a very different implementation, we need to choose a different language. We feel that re-writing the host node in C++ is the most appropriate choice of system programming language. C++ is a lower-level language than Rust and is closer to the metal. It provides greater control over system resources and over the code. This can be helpful for fine-tuning and optimizing performance, and can lead to more efficient and faster code. Furthermore, due to its long history and large user base, it is likely that we will find existing code libraries, tools, resources and examples to work with.

Other options considered tend to be fairly novel languages, or garbage collected languages, which make interoperability harder. Regarding another Rust implementation, it would be hard to come up with a different enough architecture/design to justify the project, especially given our modular approach, as we would most likely be influenced by the existing systems.

A unique approach. Despite being in C++, our proposed approach is very different to that of Soramitsu, due to our architectural vision. Indeed, our approach is modular, starting from a fork of the Parity node, and re-writing their code in C++ component by component. Consequently we should be able to have a testnet running in the early days of development, and should a component fail, it will be interchangeable with Parity's Rust crates, guaranteeing high robustness of the system.
For example, with this approach, our node will, from day zero, implement the same serialization format that Polkadot uses. This ensures that all nodes can read and process the information being sent to them in the same way, regardless of which implementation of the Polkadot host node is being used.

4.2 Project structure

As our approach in building an alternative Polkatot host node is modular, we will start from a fork of the parity host node implementation, and gradually replace crates with our own C++ implementations.

To maintain a modular structure, we will create a separate repository for all of the developed C++ code. This repository will also contain Rust crates, which will act as wrappers for the C++ code. The fork will depend directly on these crates. Our goal is for it to be as simple as possible to switch out the original implementation for our own by simply changing the dependency from the original module to our crate.

There are two approaches we can take to achieve this structure: use -sys style crates (see Section C.1), or use `cxx` (see Section C.2).

We chose to use the `cxx` crate, as it reduces the complexity of the project, allowing us to put more focus on the actual node implementation rather than the interface between the Parity node and our modules.

Of course, even with `cxx`, we need to create some code to connect the foreign function interface (FFI) with our fork of Substrate's node.

Mayon. The `Mayon` repository will contain all the C++ code, as well as the wrappers for this code, wrapping it into Rust crates, which the forked node will depend on. `Mayon`'s goal is to facilitate the integration of re-written modules into the existing node; the latter should simply depend on corresponding crates within `Mayon` in a seamless manner.

Structure. All crates associated with this project are located within the `crates` folder, with each sub-folder representing a distinct crate. Each crate is responsible for its own build process, though they should be relatively similar in structure.

The core of each crate comprises the `build.rs` and `CMakeLists.txt` files. These files are used in conjunction to enable seamless interoperability and build processes for each crate or C++ library. As mentioned above, we have decided to use `cxx` for our crates, to provide an improved experience, given that these modules are being written from scratch and will not require interfacing through a pre-existing C API. The mechanism is relatively straightforward, but certain workarounds are required to address specific issues, as detailed in Appendix C.2.1.

A template, available via ffizer, facilitates the swift creation of new crates. This template takes care of creating the necessary files and preparing the crate for the build process. To instantiate the template, it is recommended to use the `just new` command (if just is installed on your system, if not, examine the root `Justfile`). In addition, an example `hello-world` crate has been included to demonstrate how a crate can be used as a dependency. The example can be found in the `src/bin/hello_world.rs` file. It highlights that there is no need for specialized machinery to build and use the crate.

Testing. To test each C++ module that is written, we believe doctest would be a good candidate testing library. It is a lightweight and flexible testing framework that allows to write tests directly in ones source code, making it easy to test both C++ and Rust code in the same project. It is also fast, easy to use, and has good support for a variety of programming languages, including C++ and Rust. Additionally, doctest integrates seamlessly with other build systems and testing frameworks, so it can be used in a variety of different project setups. Doctest allows to write tests directly in the source code using special comments that are automatically recognized and executed by the doctest testing framework. One of the advantages of this is that it can make it

easier to write and maintain tests, as one needn't switch between source files and test files. Additionally, since the tests are included directly in the source code, it can be easier for other developers to understand and review the tests, as they can see the tests and the code being tested at the same time.

Other testing frameworks We considered were:

- Google-test.
- Catch2

Both testing frameworks are widely used, they provide a good documentation but the poor or almost null interoperability with Rust and the fact that they are not header only libraries make them not well suitable for this project.

4.3 Handling asynchrony

Managing asynchrony between the fork of the original Rust code and our C++ crates will likely present challenges. Though the C++ community's knowledge regarding the async features in C++ is sparse, our team is working towards gaining a comprehensive understanding of the key concepts and libraries that will be utilized, in order to ensure a solid foundation for the project. We also note that C++20 introduces new features in the language that makes writing asynchronous code much more natural. We further intend to incorporate C++23 features as soon as they are available in the main compilers, one of the most important for this project is std::expected. This feature is meant to facilitate error handling without the need of throwing exceptions.

Let's take a closer look at how this affects our proposed project. The Rust crate `cxx` allows calling C++ code from Rust using Rust's FFI (Foreign Function Interface). It can be used to expose C++ code to a Rust project, but it does not provide any built-in support for handling asynchronous C++ code.

To call asynchronous C++ code from Rust, we will need to write the asynchronous code in C++, and then use `cxx` to call the C++ functions that perform the asynchronous operations. The asynchrony will thus be handled at the level of C++ code. For example, using Asio, callbacks will be registered in C++ as completion handlers, and those callbacks will need to interact with Rust code through the FFI. Asio also provides other completion handlers like:

- std::future which has to be polled to get the result of an C++'s asynchronous operation. This type could be sent to Rust through a FFI opaque type, that can be "waited" on form within a Rust's async task.
- awaitable<void> handler that is meant to be used within the C++ asynchronous state machinery(C++20), not suitable to be exported to Rust using FFI.
- yield context Similar to awaitable but this blocks the async task. There seems not to be similarities with the Rust asynchronous model.

In order to make a proper assessment regarding interoperability between Rust async and C++ async paradigm, we decided to write a proof of concept, that has asynchronous Rust tasks interacting with C++ tasks using Asio. The tasks use async channels to talk to each other, this model resembles the current Polkadot design which makes use of channels also to decouple subsystems. Besides Asio, we are considering Seastar and cppcoro

libraries. It seems to be more ergonomic, its API design is quite similar to Rust, as it makes use of monads to concatenate operations.

Overall, using `cxx` to call asynchronous C++ code from Rust is possible, but it will be more challenging than using Rust's built-in `async/await` functionality, and it will require effort and care to handle the synchronization and communication between the Rust and C++ code correctly. This is due to a subtle difference between Rust async model and C++, the latter does not use a wake-up mechanism to tell the executor poll the task that is ready, so it would be necessary to add a layer between both to coordinate and connect tasks from both. We found an opinionated Rust crate called cxx-async, which is based on cxx and aims to facilitate this process.

4.4 Handling dependencies

The proposed project requires effective package management due to the numerous dependencies involved. We are looking to untangle the dependencies in the original Rust project as we are re-writing certain crates in C++, but not all of those in `paritytech/Polkadot` will be translated. Careful planning and thorough testing will be key to minimizing any issues that may arise during the process.

We have used the Cargo dependency manager to track the dependencies in Parity's project. This has helped us identify which crates depend on others, and potential issues that may arise as we begin to substitute C++ "crates" for the original Rust crates. We further have some ideas for the long term project, which we believe may help:

- Using `nix` for building the C++ project would have advantages like declarative package management, reproducibility, and isolation.
- Creating a clear mapping of which crates will be rewritten in C++ and which will not. This will help better understand the dependencies and plan out the migration process.
- When re-writing a Rust crate, and when possible, replace the dependencies with C++ version of those, so they are updated as well. This will help avoid compatibility issue with the Rust crates which are not yet translated.
- As C++ crates are substituted for Rust crates, thoroughly test the new C++ code and ensure that it's working correctly before integrating it into the main project.
- Attempt to preserve the original abstract interface for the C++ crates, so the Rust codebase does not change much and the switch is more seamless.
- Use continuous integration and continuous testing strategies to catch any issues early on and minimize the risk of encountering errors during compilation.

4.5 Keeping up to date with Parity's node

So as to ensure that modules which have been replaced stay up to date with Parity's host node. We intend to implement:

- a software application that performs automated tasks, specifically informing us of any code changes made to the relevant repository and running tests that use both the Parity modules in Rust and our implementations of modules for the host node in C++.
- tests which use only the Parity modules or only our modules. The purpose of these tests is to compare their outputs, so as to ensure they are consistent.
- integration tests using combinations of both Parity's modules and ours, to ensure that the integration of our modules in production Polkadot host is reliable and functions as intended. Tests could be scheduled to run nightly.

In addition, we will use the Polkadot forum to ask questions, and stay informed of upcoming changes to the implementation, as well as the Polkadot Protocol Proposals (RFC-style process for changes to the Polkadot Protocol), and the Polkadot Standards Proposals (describe standards for the Polkadot ecosystem), to stay up to date, and potentially suggest improvements to the system.

We would also like to suggest establishing a clear communication channel with developers at Parity to keep us informed about which branches will be merged into main. This will help us stay synchronized and prevent falling behind.

4.6 Navigating and improving the specifications

As justified in Section 3.2, the Polkadot specifications are not a sufficiently complete source of information to implement a production level host node. This being said, we have no intention of overlooking this source of information, rather the opposite. We will combine the information provided there, with help available on the Polkadot forum, as well as via the Host Implementers Element channel, so as to clarify aspects of the specifications which we feel are not up to standard. Furthermore, we will go above and beyond simply understanding the specifications, as we intend to collaborate with the Spec team from the Web3 Foundation, to bring the specifications up to date. The goal, in the long run, is to have specifications which *precede* the implementation of changes in the Polkadot ecosystem. We will work not only for detailed specifications, but also a thorough maintenance of the Polkadot Standards Proposals and Polkadot Protocol Proposals. This will benefit future alternate implementations of host nodes, thereby facilitating the establishment of a truly decentralized ecosystem for Polkadot. Additionally, high quality specifications mitigate the risks related to the loss of expertise, when key members – which are holders of crucial knowledge – leave the Polkadot ecosystem.

4.7 Translating Rust macros to C++

One strategy to handle the extensive use of Rust macros in `paritytech/Polkadot` is to start by focusing on the macros that are used in the most critical parts of the code, or the macros that have the most complex logic. Then, we can gradually work our way through the rest of the macros.

The Rust ecosystem has a tool that allows developers to *expand* a Rust macro into normal Rust code. We used this tool to expand the context bound macro that is used in all of the subsystems that comprise the Polkadot node. We thus realized that this macro defines the concrete protocol (SubsystemError and message format), and the specific subsystem it will work with. This is worth knowing as we will need to define concrete types in our FFI interface when implementing those services.

It is worth noting that C++ also has ways to perform compile-time code generation and manipulation, such as `constexpr` functions and `constexpr` variables, that we will use to achieve similar functionality as Rust macros. More details on `constexpr` functions and variables are provided in Appendix A.3.

5 Proof of concept

For our proof of concept, we have selected some components to show how our development process can be effectively used to progressively evolve the existing Parity node. This provides the opportunity to use the aforementioned libraries , while ensuring that our project structure and tooling are as useful as planned.

5.1 Selection of the replaced components

While analyzing the code stability we observed that some important crates in the Polkadot node depend on *primitives* modules. The primitives are mostly re-exports of types that are defined in Substrate but with a concrete type bound. This makes the primitives crates a good target to get an insight on how this process is going to be. We selected two primitives crates as follow:

- core-primitives
- parachain primitives

5.2 Module replacement

The core-primitives component was relatively easy to re-write as it is essentially a collection of type aliases and some re-exports with a proper type bound. We discover that type aliases are difficult as they are not really an instance but a new name for a type. In C++ there is a **using** keyword, but this is not supported by the cxx crate. This module re-export important blockchain primitives, one of them is the Header type, which is defined in substrate as follow:

Listing 1: Generic Header definition

```
1  /// Abstraction over a block header for a substrate chain.
2  #[derive(Encode, Decode, PartialEq, Eq, Clone, sp_core::RuntimeDebug, TypeInfo)]
3  #[cfg_attr(feature = "std", derive(Serialize, Deserialize))]
4  #[cfg_attr(feature = "std", serde(rename_all = "camelCase"))]
5  #[cfg_attr(feature = "std", serde(deny_unknown_fields))]
6  pub struct Header<Number: Copy + Into<U256> + TryFrom<U256>, Hash: HashT> {
7          /// The parent hash.
8          pub parent_hash: Hash::Output,
9          /// The block number.
10         #[cfg_attr(
11                 feature = "std",
12                 serde(serialize_with = "serialize_number",
13                       deserialize_with = "deserialize_number")
```

```
14          )]
15          #[codec(compact)]
16          pub number: Number,
17          /// The state trie merkle root
18          pub state_root: Hash::Output,
19          /// The merkle root of the extrinsics.
20          pub extrinsics_root: Hash::Output,
21          /// A chain-specific digest of data useful for light clients
22          /// or referencing auxiliary data.
23          pub digest: Digest,
24  }
```

We can see that this type definition is rather generic, which means that this definition should work with any hash protocol, Number width as long as those types meet the restrictions imposed by the trait boundaries. The type is re-exported in the runtime module, but this time, it is not longer generic:

Listing 2: Header re-export in Rust

```
1  pub struct Header {
2      pub parent_hash: H256,
3      pub number: u32,
4      pub state_root: H256,
5      pub extrinsics_root: H256,
6      pub digest: sp_runtime::generic::Digest,
7  }
```

We use a *u32* as the type for the Blocknumber and *BlakeTwo256* as the Hash protocol. This is optimal as any type defined in Rust has to be interfaced with C++ types supported by the **cxx** crate, otherwise the type has to be declared as an opaque type. In C++ this type could be defined as:

Listing 3: Header representation in C++

```
1  struct Header {
2          pub parent_hash: [u8; 32],
3          pub number: u32,
4          pub state_root: [u8; 32],
5          pub extrinsics_root: [u8; 32],
6          pub digest: Vec<u8>,
7  }
```

The *Hash::Output* type for *BlakeTwo256* is represented as an array of 32-bytes.

Another core-primitive type define in this module is:

Listing 4: Candidate Hash

```
1  /// A hash of some data used by the relay chain.
```

```
2 pub type Hash = sp_core::H256;
3
4 /// This type makes it easy to enforce that a hash is a candidate hash on the type
      ↪ level.
5 #[derive(Clone, Copy, Encode, Decode, Hash, Eq, PartialEq, Default, PartialOrd, Ord,
      ↪ TypeInfo)]
6 pub struct CandidateHash(pub Hash);
```

Here sp_core::H256 is just a wrapper around an array of 32 bytes. This is represented in C++ as follow:

Listing 5: CandidateHash representation in C++

```
1 struct CandidateHash {
2         pub hash: [u8; 32],
3 }
```

A inquiring observer would notice a subtle difference between the Rust definition and the C++ one, The reason is due to the fact that C++ does not support tuple structs, This is a limitation when writing compatible code for Rust.

The

parachain/primitives module is more complex. It defines various structs, though, many of them are just wrappers around Rust's primitives types like integers.

Our current test re-write – the core-primitives and parachain repositories in `mayon/crates` – mostly interfaces some types to C++, so that other C++ modules can import them. We believe that it is better to implement an FFI layer which interfaces with the definitions of imported types. This implies that at some point, we will need to re-write the most used substrate modules and include them in our code base. As an experiment, we started to do this for the sp-primitives/runtime module. The objective was to define generic types for some elements that the core-primitives re-exports. This re-write is not expected to cover all the modules from a substrate dependency but the ones that are required by the primitives definitions in Polkadot.

Substrate design separates data from functionality. The described **Header** type is defined as a struct, but there is also a Header trait that defines methods/functionality that any instance of Header struct must provide. This decoupling from the concrete type facilitates our implementation making it clear what are the methods our Header instance provides, and that there would no be compilation errors from Rust due to differences between different instances.

5.3 Integration with a fork of the Polkadot node (Parity)

This Proof of concept has been integrated into Polkadot in our experimental branch. It allowed us to verify that:

- Compilation works.

- The node runs properly.
- Polkadot tests pass.

We decided to leave the re-written modules in the repo but, out of the compilation tree, in the workspace. This could be useful to track upstream changes or updates, and as a roll-back, helping us to ensure that our hybrid node would be always online even if we detect that something went wrong with any of our modules, in that case, we re-build the node using this time, the native rust implementation for that module.

5.4 Reproducible builds with nix

When we reached the proof of concept phase, we had to determine how we wanted the alternative modules to be integrated into the host. Considering that Polkadot is a Rust project, making sure the crates are easy to use with cargo would render them ergonomic and idiomatic, greatly simplifying dependency management. For this reason, we had initially investigated having a build script encapsulate the C++ build phase, which would be using CMake and Hunter (for package management), thus the crate could be easily be injected at a dependency without having to substantially change how the Polkadot Node is built, but rather just specifying a different source for a given crate.

With Hunter, one can manage dependencies for CMake projects and automatically fetch the dependencies one need for a project. Indeed, when using Hunter, one specifies the version of Hunter one wants to use in the CMake project. Hunter then uses this information to determine the correct versions of the dependencies needed, and fetches them. This eliminates the need to manually download and manage dependencies.

Another option we looked into was using Nix to handle the dependencies and build of the C++ libraries. An application to `Mayon` can be found in the `nix` repository.

Nix is a powerful package manager that offers several benefits for managing project dependencies. It ensures reproducible builds, allows for easy creation of isolated development environments and can be used for different programming languages, including C++ and Rust. Its atomic upgrades and rollbacks, along with its declarative configuration format, make it a reliable and convenient tool for managing dependencies.

With Nix, we would be using the `crane` library to package the Rust project and pull in the required dependencies for a given crate, and the `fenix` library to get the correct Rust version. We have looked over alternative tools such as `naersk` and `Rust-overlay`.

Due to the nature of the libraries used, some packages were not readily available in Nix and we had to package them ourselves.

To this end, between the many options, we adopted `niv`. This allowed us to keep the repository's flake clean, whilst making it easy to to add a new source, to build a specific package for the project.

The issue with using Nix is that the build encapsulation we had previously developed can not be easily translated, so we came up with 2 different solutions:

1. *Do it the Nix way, and have Nix manage the build completely.* This means supporting the build of the Polkadot Host entirely via Nix, by creating a Flake for the Host and pulling in the required dependencies (most directly by specifying the project's repository as input to retrieve the package with its dependencies). The advantage here is that we would have more control over the build process and the involved dependencies, but it would require users to at least get slightly familiar with Nix.

2. *Do it the Rust way, and have Cargo manage the build.* This has the same advantages as CMake + Hunter, and Nix would be invoked by the build script to pull in the dependencies for the C++ library, then build the library and make it available for linking by `Rustc`. One disadvantage with this approach is that it shifts the requirement from having cmake available to having nix available on the machine, with the latter being much less popular. It's worth mentioning that Nix portable exists, which would allow us to pull in nix as required during the build phase, but we do not have high confidence in this.

Another option exists, and that would be to support both: if Nix is available on the system then use it to pull the dependencies and manage a given library's build, so cargo can link it afterwards. If not, then fallback to a different build system, like CMake + Hunter, to fetch the dependencies and build the project. Of course, supporting both builds will require more effort, and we might find incompatibilities between the same dependencies pulled one way or the other. However, with proper planning and testing, it should be possible to implement this approach and provide flexible yet robust builds for the Polkadot Host node.

5.5 Lessons learned

There are multiple difficulties we were not aware of before working on this proof-of-concept:

cxx crate limitations

1. **Custom derives are not supported** The parachain-primitives define the type Id, which is just a wrapper around an integer. It is defined as follows:

Listing 6: Header re-export in Rust

```
/// Unique identifier of a parachain.
#[derive(
    Clone,
    CompactAs,
    Copy,
    Decode,
    Default,
    Encode,
    Eq,
    Hash,
    MaxEncodedLen,
    Ord,
    PartialEq,
    PartialOrd,
    RuntimeDebug,
    TypeInfo,
)]
#[cfg_attr(
    feature = "std",
```

```
20      derive(serde::Serialize, serde::Deserialize, derive_more::Display)
21  )]
22  pub struct Id(u32);
```

Although, rather simple, What makes this type complex is not the number of *Traits* It implements but the non supported traits it requires. The **cxx** crate only supports "standard" trait derives, leading to an error if any of the custom traits declared in this definition are added to the C++ definition. There are two solutions to this problems:

- The type is declared in Rust as an opaque type.
- Define it as a shareable type so Rust and C++ have the same definition.

We came-up with the second solution, however, because our type and the original Id have a different layout, We need to add methods to convert From/To each other, as follows:

Listing 7: Conversion between Rust/C++ types

```
1  impl From<crate::primitives::Id> for ffi::Id {
2      fn from(value: crate::primitives::Id) -> Self {
3          ffi::Id { id: value.into() }
4      }
5  }
6
7  impl From<ffi::Id> for crate::primitives::Id {
8      fn from(value: ffi::Id) -> Self {
9          Self::from(value.id)
10     }
11 }
```

This is also necessary because of the next limitation.

2. **Lack of support for tuple structs**, We already mentioned this before. C++ does not support tuple structs, so we declare them as a struct with one field. This requires implementing the From trait for the Rust and C++ types.

3. **Only support for simple enum types**, Although C++20 added support for *sum types*, the cxx crate does not support exposing Rust enums to C++, this is a problem that can be solved by defining each enum variant as structs in the C++ side, then, we define an opaque type in C++ that would represent a full std::variant type export proper methods to convert the Rust enum to this opaque type.

4. **Wasm compilation** The wasm support in C++ is not as mature as Rust. The recommended target for wasm in C++ is wasm32-unknown-emscripten, however the target in Polkadot is wasm32-unknow-unknown which we found are not full compatible. The modules in Polkadot that are also compile to wasm are:

    - core-primitives

- primitives
- parachain

We decided that for those crates we omit cross-compilation to wasm for the C++ modules, as our re-write is meant to be for the Polkadot host implementation.

API compatibility: The changes in one module should pass undetectable to other Rust modules that use a well defined API definition for each primitives. Regardless if a type is fully shareable, it is likely that it can not be defined as it is due to the fact that it might need to derive custom traits, and those traits could be the only thing other Polkadot components known about the type.

Including new libraries: For those unfamiliar with Nix, incorporating new libraries is a challenging task. This raises the question of finding a robust build system that enables the seamless inclusion and removal of dependencies. We emphasize the importance of making the build system more manageable, as we will need to integrate numerous C++ libraries in the future.

5.6 Unit testing

At the moment, our C++ code is compiled as part of the Rust compilation step. Attempting to compile the C++ code directly for testing will result in failure, as the 'cxx' crate generates header files that contain type definitions required by the C++ code. So as to ensure that unit testing the Rust code will test the C++ code, we will write our unit tests as special Rust code that calls the Rust-C++ modules we want to test. A similar approach is used in Substrate, and we believe this should work effectively. Precisely, we will have a Rust layer that interacts with C++, and it is the layer that our unit tests use.

There may also be some C++ code that is not used by the FFI interface, but is used in our C++ implementation. The C++ code in question may include core functions such as (de)serialization, encoding and decoding, getting hashes, performing verification, and performing type conversions. This code can be tested directly using C++ unit testing using the doctests library which also provides compatibility with Rust, moreover, a make rule can be created to run the tests.

5.7 Integration tests

As our proof of concept is built upon a fork of the `paritytech/Polkadot` repository, we can run the test suite defined in Parity's repository. So far running Parity's cargo tests on our hybrid node is successful.

As the code-base evolves, one should use our already defined C++ testing framework, and add additional unit tests while writing C++ code.

5.8 Change management strategy

In order to keep up to date with Parity's code base, we can update our fork by rebasing. To reduce conflicts, we advise to perform changes in a different file. For example: `primitives_ffi.rs` would contain `cxx` and `ffi` definitions, `primitives.rs` would be left untouched, but one should update `lib.rs` to export our types. Of course, at some point, the untouched `primitives.rs` will not longer be necessary.

The same applies for Substrate crates. For example we added the `sp-primitives` folder in our proof of concept, it contains the `runtime` crate, which at the moment only exports the `Header` type. The latter is no longer generic, but uses concrete types that Polkadot uses. We simply copied the definition, removed the generics to use concrete types, added the FFI, and some functions, and exported the lot in `runtime/src/libs.rs`.

Listing 8: `runtime/src/lib.rs`

```
1 #![cfg_attr(not(feature = "std"), no_std)]
2 #![warn(unused_crate_dependencies)]
3
4 mod header;
5 pub use header::Header;
```

Note that we do not have a way to track changes on the `Header` type, as it is a custom copy of the generic type defined in the original repository. In fact, our re-writes use this custom `sp-primitives/runtime` to use the `Header` type, but use the `substrate/sp-primitives/runtime` to use the `Header` Trait, that defines the behavior of the `Header` struct.

6 Conclusion

In conclusion, the task of rewriting the Polkadot host node in C++ using a hybrid approach is a challenging but exciting opportunity. Despite the complexity of the original project's dependencies, the difficulties of asynchronous programming in C++, and the use of Rust macros in the Parity project's codebase, we are confident that with careful planning and a modular approach, we can successfully complete this project.

Although the reliability of the specification documents is not entirely sufficient, we are fortunate to have access to the wealth of knowledge from the members of the Web3 Foundation and the Parity developer community. By leveraging their expertise, we can overcome any obstacles and create a reliable and efficient Polkadot host node in C++.

It's important to note that this project is not to be taken lightly, and we understand the importance of having a stable team with diverse skills and reliable communication channels between the Web3 Foundation, Parity, and our development team. However, with the right resources and a positive attitude, we are confident in achieving great success in this project. Overall, we are excited about the potential of this project and believe that it will have a positive impact on the Polkadot ecosystem.

Appendix A Risk identification and management – Details

A.1 Detailed analysis of the Web3 Foundation's specifications

A.1.1 Gaps in the technical specification

Various algorithms/definitions/entire sections remain empty. It is not clear – from the specifications – how these undefined aspects of the host node should be implemented. For example, as of January 3rd 2023:

- The algorithm for maintaining the transaction pool is not defined.

- Runtime compression, disputes, Erasure encoding are not documented.
- Storage entry modifier might be incorrect and should be reviewed.
- The section specifying the cryptographic algorithm used to generate randomness is empty.
- Various subsections relating to cryptographic keys are absent. In particular:
  - holding and staking funds,
  - creating a controller key,
  - designating a proxy for voting, and
  - controller settings.
- There are a number of TODOs throughout the specifications for e.g. missing references, definitions and detailed data structures.

A.1.2 Errors in the technical specifications

We here list some errors in definitions and poorly defined definitions, where these definitions are specific to the Polkadot host node:

- In the definition of a node header, the definition of the length of the partial key pk_N^l depends on indices x and y which are not defined.
- In the definition of a node key, we believe the partial key pk_N should be indexed from k_{i+1} rather than k_i, and KeyEncode(k) should be equal to $(k_{enc_1}, \ldots, k_{enc_i}, k_{enc_{i+1}}, \ldots, k_{enc_{i+l_{pk_N}}})$, where $pk_N^{Aggr} = (k_{enc_1}, \ldots, k_{enc_i})$.
- In the definition of the index function, we believe the mentions of k_N should in fact be $\mathsf{KeyEncode}(k_N)$.
- In the aggregate-key algorithm, the index i is not clearly incremented. Furthermore the aggregate public key of a child is defined from its parents' key $k_{N_{parent}}$, the index $\mathsf{Index}_{N_{parent}}(N_{child})$ and its own public key $pk_{N_{child}}$. However, it is not clear to us how one should compute $\mathsf{Index}_{N_{parent}}(N_{child})$, since computing the index function (according to its definition) requires knowing the child's node key and partial key.

A.1.3 W3f test suite for the host node

We here compare the test coverage of the host node from Polkadot-tests, with the specification documents. The hostapi-runtime module contains the host API tests which will call the Polkadot host implementation for testing purposes. The test file is divided into subcategories:

State Trie Tests the functionality of the Trie API in the Polkadot host node. There are two main tests that can be run: `trie-root` and `insert-and-delete`. The `trie-root` test computes the root hash of a trie based on a set of key-value pairs read from a yaml state file. The `insert-and-delete` test creates a trie from a set of key-value pairs, inserts and deletes additional key-value pairs, and checks that the trie has the expected root hash after these operations. There are also helper functions for reading the key-value pairs from the state file and for converting the keys and values to and from hex encoded strings.

Scale Codec Tests related to the Scale Codec feature of the Polkadot host node API. The Scale Codec is a library that provides encoding and decoding of data in the SCALE format which is used in Polkadot.

Storage API Tests the functionality of the Polkadot host node's storage API. They include tests for setting and getting key/value pairs in storage, reading values from storage, clearing values from storage, and checking if keys exist in storage. The tests also compute and print the storage root when they are initialized.

Default child storage API These tests cover setting, getting, reading, and clearing key-value pairs in a child storage namespace. The tests take parsed input and a runtime object as arguments and use the runtime to call functions with the specified input. The tests check that the correct output is returned for different input combinations and that the values stored in the child storage are correct.

Crypto API Tests the Polkadot host node's crypto API. The following functionalities are tested:

- **Ed25519** Generating and retrieving a list of ed25519 public keys; generating an ed25519 key pair; signing a message with an ed25519 key pair; verifying the signature of a message with an ed25519 public key.
- **Sr25519** Generating and retrieving a list of sr25519 public keys; generating an sr25519 key pair; signing a message with an sr25519 key pair; verifying the signature of a message with an sr25519 public key.
- **Hashing** a message.
- **Encrypting and decrypting** a message with a key pair.
- **Encrypting and decrypting** a message with a shared secret.

Hashing API Tests the functionality of the Polkadot host node's hashing API. The tests take a runtime instance and a string representing the name of a hashing function (e.g., "keccak-256"), as well as a ParsedInput containing the data to be hashed. The function then calls the appropriate hashing function on the input data and prints the resulting hash in hexadecimal form.

Allocator API Tests the allocation and freeing of buffers. One test checks that a buffer can be allocated and its content is returned correctly, while the other test checks that a previously allocated buffer can be successfully freed.

Offchain API Tests various offchain functionalities such as checking if the current node is a validator, submitting transactions, retrieving network state and timestamp information, sleeping until a certain point in time, generating random seeds, and setting and getting values from local storage.

Genesis default (resp. legacy) The test will check that the genesis default configuration (v1) (resp. legacy configuration (v0)) is working as expected by checking that the expected calls and hashes are returned by the host node. It also checks that the grandpa and babe configurations are requested. This test is to check that the host node is able to start with the proper configuration and that it is able to make the expected calls and return the expected hashes.

Many parts of the Polkadot specification for the host node do not appear to be covered by tests in Polkadot-tests. A few examples of untested requirements: discovery mechanisms (for finding peers within the network),

block announcements and requests, warp sync protocols, equivocation[2], and many more. Generally, there does not seem to be any tests ensuring that the different Polkadot implementations can correctly communicate with each other.

A.2 Discrepancies in listed network protocols

There is a list of protocols in the Polkadot specifications, along with a description of how the protocol specifiers are constructed. Protocols in that list which are not part of the current Parity node implementation:

- `/ipfs/id/1.0.0`
- `/dot/kad`
- `/dot/light/2`
- `/91b171bb158e2d3848fa23a9f1c25182fb8e20313b2c1eb49219da7a70ce90c3/block-announces/1`. Note: the specs do not say whether this hash is the PeerId, but the implementation uses the genesis block hash and the fork_id (if available).

Remark 1. *It is possible that some protocols in the specifications are part of the `libp2p-Rust` internals or the substrate network library. The latter relies on the `NetworkConfig` module which – according to the documentation in substrate – holds a list of protocols (streams) that the node should open so that the protocol name can be part of a configuration file.*

Protocols in Parity's node implementation which are not in the specifications:

- /req_chunk/1
- /req_collation/1
- /req_pov/1
- /req_available_data/1
- /req_statement/1
- /send_dispute/1

Remark 2. *The names in the list above are legacy protocol names associated with each peer set. The number "1" in the specs refers to legacy support.*

A.3 `constexpr` functions and variables

`constexpr` functions and variables were introduced in C++11. They were added to provide a way to perform computation at compile-time, which allows for more efficient and optimized code. The `constexpr` keyword is used to indicate that a function or variable can be evaluated at compile-time, if all of its arguments are known at compile-time.

[2]The Polkadot Host must detect if a voter broadcasts two or more valid votes to blocks during one voting sub-round, and submit it to the Runtime

More precisely, a `constexpr` function is a function that can be evaluated at compile-time if all of its arguments are known at compile-time. This allows the compiler to perform the computation during compilation, and the resulting value can be used in a `constexpr` context, such as initializing a `constexpr` variable. Similarly, `constexpr` variables are variables that are evaluated at compile-time and have the same value in all translation units and throughout the program's lifetime. They are similar to `const` variables, but their value can be computed at compile-time.

In C++11, `constexpr` functions could only contain a single return statement and no loops or other control flow statements. C++14 relaxed some of those restrictions, allowing loops and more complex computation in `constexpr` functions. C++17 and C++20 introduced even more capabilities to `constexpr`, such as if `constexpr`, template argument deduction, and more.

Appendix B Code analysis

In this appendix we sift through the `paritytech/Polkadot` code base so as to have a clear view of the internal dependencies between crates of the project (referred to as *Polkadot Dependencies*), and dependencies on crates from Substrate.

We also evaluate the stability of the crates which constitute Parity's host node. A crate is considered stable after 3 months without changes or, failing that, if changes were relevant neither in terms of design or of functionality.

Client module

The `Polkadot-client` crate has a large number of dependencies, including dependencies on the Polkadot runtime, as detailed in Table 1.

This crate provides an interface for interacting with the runtime of the Polkadot network. The crate is intended to be used by both the host node and the runtime itself. It provides a set of traits and types that can be used to make calls to the runtime, query the state of the blockchain, and subscribe to events. The Client enum combines various implementations of the client interface for different runtimes, such as Polkadot, Kusama, Rococo, and Westend. The Client enum and its associated types can be used by both the host node and the runtime to make calls to the runtime, query the state of the blockchain, and subscribe to events. The Polkadot-client crate also provides a set of utility functions and types that are used to facilitate testing of the runtime code, such as mock data providers and types for testing asynchronous code.

Component	Substrate Dependencies	Polkadot Dependencies	Stability	Notes
Polkadot-client	frame-benchmarking frame-benchmarking-cli pallet-transaction-payment pallet-transaction-payment-rpc-runtime-api frame-system frame-system-rpc-runtime-api sp-consensus sp-storage sp-runtime sp-blockchain sp-api sp-core sp-keyring sp-inherents sp-timestamp sp-session sp-authority-discovery sp-finality-grandpa sp-consensus-babe sp-transaction-pool sp-offchain sp-block-builder sp-mmr-primitives sc-consensus sc-executor sc-client-api sc-service beefy-primitives	#Runtimes Polkadot-runtime kusama-runtime westend-runtime rococo-runtime # Polkadot-core-primitives primitives node-core-parachains-inherent runtime-common	Unstable	Changes every ∼ 2 weeks

A crate is considered stable after 3 months without changes or, failing that, if changes were relevant neither in terms of design or of functionality.

Table 1: Dependencies & stability of `Polkadot-client`.

Collation generation module

The collation generation subsystem is the interface between Polkadot and the collators. Collators are parachain nodes which produce candidate proposals and send them to the relay chain validator.

Dependencies and stability of the `Polkadot-node-collation-generation` crate are detailed in Table 2.

Component	Substrate Dependencies	Internal Node libraries		Stability
collations	sp-core sp-maybe-compressed-blob	primitives node-primitives erasure-coding	subsystem subsystem-util gum	Stable

Stable = 3 months without changes, or changes irrelevant in terms of design/functionality.

Table 2: Collation generation dependencies and stability

Core modules

The core module contains different components, some are independent of the specification and represent Parity's architectural approach to fulfill the required functionality for modules whose behavior is defined in the specification. Core behaviors relate to internal work that a specific node does. The core module often interacts with the network module, but they can be heavily abstracted from each other. Core behaviors care that information is distributed and received, but not the internal details of how distribution and receipt function.

Table 3 details the core submodules and their dependencies, and Table 4 details their stability.

Component	Stability	Notes
approval-voting	Unstable	Regular changes, though many are minor.
av-store	Unstable	
backing	Stable	
bitfield-signing	Stable	Minor changes last November.
candidate-validation	Partially stable	Recent changes relative to error-handling. No important structural/functional changes.
chain-api	Stable	
chain-selection	Stable	
dispute-coordinator	Unstable	Significant changes.
parachain-inherent	Unstable	
provisioner	Unstable	Regular changes, though most are minor.
pvf-checker	Stable	Minor changes in January.
pvf	Unstable	Regular changes, though many are minor.
runtime-api	Partially stable	Changes to `runtime-api` cache in November.

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 4: Core submodules stability.

Gum module

The tracing-gum crate does not directly rely on any Substrate dependencies. It is not present in any specifications, as it simply provides additional features to the tracing crate. It is designed to be used in conjunction with mick-jaeger, a crate for distributed tracing, in order to allow logs and spans to be cross-referenced in Grafana. It does this by adding a shared traceID or TraceIdentifier to logs that can be used to link them to spans in mick-jaeger. The crate includes a proc-macro that allows developers to easily add this traceID to their logs by simply using a modified version of the `tracing::warn,info,debug,trace` macros, rather than manually adding the traceID to each log.

Component	Polkadot Dependencies	Stability	Notes
tracing-gum	Polkadot-node-jaeger gum-proc-macro Polkadot-primitives	Stable	The `gum-proc-macro` crate does not depend on any Substrate or Polkatot crates.

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 5: Dependencies & stability of the Gum module.

Jaeger module

The `Polkadot-node-jaeger` crate is not mentioned in the Polkadot specs, as it is a tool to ensure the node runs smoothly. The `Polkadot-node-jaeger` provides primitives used by Polkadot for interfacing with Jaeger. Jaeger is a distributed tracing system that allows to monitor and troubleshoot issues in an application. It helps understand how different components of an application are interacting and can be used to identify performance bottlenecks and errors. In the context of Parity's implementation of the Polkadot host node, Jaeger may be used to trace transactions as they pass through the node and to monitor the performance of the node itself.

Malus module

The `Polkadot-test-malus` crate is specific to Parity's implementation (not a requirement of the Polkadot specifications). It is used for testing purposes, as it allows to create misbehaving nodes for local testnets, system and Simnet tests. It has many Polkadot dependencies, and though it only *directly* depends on the substrate crates `sp-keystore` and `sp-core`, it also relies on `Polkadot-client` (Appendix B), which heavily depends on substrate crates.

Metrics

The `Polkadot-node-metrics` crate is not a requirement of the Polkadot specifications. It provides helpers for collecting metrics. This crate also reexports Prometheus[3] metric types which are expected to be implemented

[3]Prometheus is one of the most widely used monitoring tools for managing highly available services supported by Cloud Native Computing Foundation. By providing Prometheus metrics in Substrate, node operators can easily adopt widely used display/alert

by subsystems. See Table 6 for details on its' stability and dependencies.

Component	Substrate Dependencies	Polkadot Polkadot	Stability	Notes
Polkadot-node-metrics	sc-service sc-cli substrate-prometheus-endpoint sc-tracing	gum primitives	Stable	

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 6: Dependencies & stability of `Polkadot-node-metrics`.

Network modules

The network module contains different components, some are independent of the specification and represent Parity's architectural approach to fulfill the required functionality for modules whose behavior is defined in the specification. The Network module is particularly complex as it makes use of Rust asynchronous features, which will cause difficulties when interfacing asynchronous Rust code with C++.

In Table 7, we detail the submodules in the network module, along with their dependencies, and in Table 8 we expose their stability.

tools such as Grafana and Alertmanager. Easy access to such monitoring tools will benefit parachain developers/operators and validators to have much higher availability of their services.

Component	Substrate Dependencies	Internal Node libraries
protocol	sc-network sc-network-common sc-authority-discovery	primitives node-primitives node-jaeger
gossip-support	sp-application-crypto sp-keystore sp-core sc-network	protocol
statement-distribution	sp-staking sp-keystore	node-primitives subsystem subsystem-util protocol
collator-protocol	sp-core sp-runtime sp-keystore	primitives node-primitives subsystem subsystem-util protocol
bridge	sc-network sc-network-common sp-consensus	primitives overseer subsystem subsystem-util protocol
approval-distribution		primitives node-primitives subsystem subsystem-util protocol
availability-distribution	sp-core sp-keystore	primitives node-primitives erasure-coding subsystem subsystem-util protocol
availability-recovery	sc-network	primitives node-primitives erasure-coding subsystem subsystem-util protocol
bitfield-distribution		primitives subsystem subsystem-util protocol
dispute-distribution	sc-network sp-keystore sp-application-crypto	primitives node-primitives erasure-coding subsystem subsystem-util protocol

Table 7: Network submodules and their dependencies

Component	Stability	Notes
protocol	Stable	Heavy dependencies on Rust asynchronous programming. Heavy dependencies on Rust asynchronous features. Candidate to changes regarding new protocol definitions.
gossip-protocol	Stable	A component with only one file.
statement-distribution	Stable	Cosmetic changes to improve API. some files have not been changed since a year ago.
collator-protocol	Unstable	Big changes 2 months ago.
bridge	Unstable	Direct dependency on overseer module
approval-distribution	Partially stable	lib.rs changed 2 months ago with refactoring updates
availability-distribution	Stable	Minimal change two months ago
availability-recovery	Stable	Typos fix two months ago
bitfield-distribution	Stable	
dispute-distribution	Stable	The last change 1 month ago has to do with a dependency API update

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 8: Network submodules change ratio

Overseer module

The `Polkadot-overseer` crate implements the Overseer architecture described in Paritytechs' host implementer guide. Though there is no explicit mention of the overseer in the Polkadot specifications, the overseer is responsible for crucial roles of the Polkadot host such as importing and finalizing blocks.

Furthermore, it allows spawning, stopping and overseeing asynchronous tasks as well as establishing a well-defined and easy to use protocol that the tasks can use to communicate with each other.

See Table 9 for details on its' stability and dependencies.

Component	Substrate Dependencies	Polkadot Dependencies	Stability
Polkadot-overseer	sc-client-api sp-api sp-core	# Polkadot- node-network-protocol node-primitives node-subsystem-types node-metrics primitives tracing-gum	**Stable** (some refactoring in 12/2022)

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 9: Dependencies & stability of overseer

.

Primitives modules

There are different categories of primitives across the repository:

- Core primitives,
- Polkadot primitives,
- Node primitives.

Most of these 'primitives' submodules depend on primitive types defined in the Substrate project. Primitive types are intended to define basic elements that encapsulate common functionality and that are used as building blocks for more complex constructions used for component-specific functionality.

The dependencies and stability of all three categories of primitives are detailed in Table 10.

Component	Substrate Dependencies	Polkadot Dependencies	Stability	Notes
Core primitives	sp-core sp-std sp-runtime		Stable	non structural changes
Polkadot primitives	sp-api sp-arithmetic sp-authority-discovery sp-consensus-slots sp-io sp-keystore sp-staking sp-std sp-runtime sp-inherents sp-application-crypto	core-primitives Polkadot-parachain	Partially stable	Recent change included the removal of a substrate's dependency, this did not require structural changes in previous code.
Node primitives	sp-core sp-application-crypto sp-consensus-vrf sp-consensus-babe sp-keystore sp-maybe-compressed-blob	Polkadot-primitives Polkadot-parachain	Partially Stable	A new status module was added recently whereas the remaining code is fairly stable. Although the usage of async makes it difficult to rewrite at an early stage of development.

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 10: Dependencies & stability of all primitive categories.

Service module

The `Polkadot-service` crate provides a specialized wrapper around the Substrate service. It defines various types and functions related to the operation of the host node, including types and functions for managing the chain, handling RPC requests, and interacting with the network.

The `Polkadot-service` crate also includes a number of submodules that provide additional functionality, such as the `chain_spec` module, which provides types and functions for defining and working with chain specifications, and the `grandpa_support` module, which provides support for the GRANDPA finality gadget.

In general, it seems that the `Polkadot-service` crate provides a number of types and functions that are used to manage and coordinate the various components of the Polkadot host node, and it helps to ensure that the node is able to operate effectively and support the needs of the network.

The `Polkadot-service` crate heavily relies on substrate (24 dependencies on substrate client, 19 on substrate

primitives, 4 on substrate pallets and 2 other substrate dependencies), and on other Polkadot components, including the **Polkadot runtime**.

Subsystem-test-helpers

The `Polkadot-node-subsystem-test-helpers` crate is a collection of utility functions and types that are used to facilitate testing of subsystems in the Polkadot project. As such it is not part of the Polkadot host node specifications. It provides various mock data providers and other utility functions that can be used to test the Overseer, the subsystems and the tasks that they spawn. It also provides types like `SingleItemSink` and `SingleItemStream`, which are used to test asynchronous code that involves sending and receiving values over a sink and stream.

Subsystem-types

The `Polkadot-node-subsystem-types` crate is a collection of types and definitions used by the subsystems in a Polkadot node. It provides a common set of types and definitions that can be used by different subsystems to communicate and coordinate with each other. These types include messages that can be passed between subsystems, as well as types for storing and tracking the state of various components in the node, such as the active leaves of the relay chain. The crate also provides some utility functions and macros that can be used by subsystems.

Component	Substrate Dependencies	Polkadot Dependencies	Stability	Notes
Polkadot-node-subsystem-types	sc-network sp-api sp-consensus-babe sp-authority-discovery substrate-prometheus-endpoint	# polkatot-primitives node-primitives protocol node-jaeger statement-table	Partially stable	Recent changes are for removing unused code / refactoring or improving logging & error handling.

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/-functionality.

Table 11: Dependencies & stability of `Polkadot-node-subsystem-types`.

Subsystem-util

The `Polkadot-node-subsystem-util` crate is a utility crate for use by subsystems in the Polkadot node. It provides various common functions and types that may be useful to subsystems, such as a utility for canceling a group of spawned jobs and a utility for determining the local node's validator ID. It also reexports Prometheus metric types that subsystems may implement and use.

See Table 12 for dependencies and stability of this component.

Component	Substrate Dependencies	Polkadot Dependencies	Stability	Notes
Polkadot-node-subsystem-util	sp-core sp-application-crypto sp-keystore	tracing-gum Polkadot-overseer Polkadot-primitives # Polkadot-node-subsystem jaeger metrics network-protocol primitives	Partially stable	Most changes are minor.

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/-functionality.

Table 12: Dependencies & stability of `Polkadot-node-subsystem-util`.

Subsystem module

The `Polkadot-node-subsystem` crate provides various types and functions that are used to manage and coordinate the various subsystems that make up the host node.

The crate exports a number of types and functions from the `jaeger` and `overseer` crates, which are used for tracing and coordinating the subsystems, respectively. It also defines a number of types and functions specific to the subsystems of the Polkadot host node.

It does not directly depend on any Substrate crates.

Component	Polkadot Dependencies	Stability	Notes
Polkadot-node-subsystem	# Polkadot-overseer node-subsystem-types node-jaeger	Stable	

A crate is considered stable after 3 months without changes, or if changes were irrelevant in terms of design/functionality.

Table 13: Dependencies & stability of `Polkadot-node-subsystem`.

Zombienet-backchannel

The `zombienet-backchannel` crate is not part of the w3f's Polkadot-spec. We understand that it is a component of the Parity implementation of the Polkadot host node that provides a mechanism for coordination between malicious actors and the zombienet test-runner (zombienet is a testing framework for Substrate-based

blockchains that allows users to easily create and test ephemeral networks). The `zombienet-backchannel` crate appears to implement a 'backchannel' using a WebSocket connection that allows the test runner to send messages to and receive messages from runtime instances being tested. In particular, the `zombienet-backchannel` crate provides the possibility to test the behavior of the host node in different scenarios, including potentially testing the behavior of the node when faced with malicious actors.

The crate defines a number of types and functions for establishing and interacting with the WebSocket connection, including the ZombienetBackchannel, Broadcaster, and BackchannelItem types. The Broadcaster type provides a simple API for sending messages through the backchannel, and the ZombienetBackchannel type handles the underlying WebSocket connection and message passing.

Appendix C Details for choice of project structure

As our approach in building an alternative Polkatot host node is modular, we will start from a fork of the parity host node implementation, and gradually replace components with our own C++ implementations. Our goal regarding the project structure is to have a repository – separate from the fork – which contains all the Zondax C++ code. This repository will also contain crates, acting as wrappers for the C++ code, on which the fork will depend directly. (ideally it would be as simple as changing one of the modules to point to our crate instead).

To achieve such a structure, we see two possible approaches: use -sys style crates (see Section C.1), or use `cxx` (see Section C.2).

C.1 -sys style crate

In the Rust programming language, a "-sys style crate" refers to a crate (a package of Rust code) that provides an interface to a native (non-Rust) library or system. These crates are often used to provide Rust bindings to C or C++ libraries, and they are typically named with the "-sys" suffix to indicate that they are system-level crates. -sys style crates are typically used when there is no existing Rust crate available for the library or system you want to use, or when you want to provide a higher-level or more idiomatic interface to the library or system. They can also be used to wrap native libraries that do not have a stable API, or that have a different API on different platforms. -sys style crates are typically implemented using Rust's foreign function interface (FFI) and are designed to be as lightweight as possible, with minimal abstractions over the underlying native library. This can make them somewhat more difficult to use than higher-level Rust crates, but it also allows them to offer more fine-grained control and better performance in some cases. Usually -sys crates also specify a links entry in their manifest, telling cargo that the crate requires an external dependency and shouldn't be duplicated in the build tree, thus no multiple version of a -sys crate are possible (normally) in the dependency tree.

C.2 cxx

`cxx` is a Rust crate that provides a safe and easy-to-use interface for calling C++ code from Rust. It allows one to write Rust code that calls C++ functions, classes, and methods, and it handles the necessary type conversions and memory management automatically. `cxx` can be a useful alternative to using a -sys style crate

when you need to interact with C++ code from Rust. It provides a higher-level and more idiomatic interface than -sys style crates, which can make it easier to use and more maintainable in some cases. Using `cxx` does require that you have a C++ compiler installed on your system, and it may not be as lightweight or performant as a -sys style crate in some cases. However, it can be a good choice if you are more familiar with C++ than with Rust's FFI, or if you want to write code where large or complex C++ and Rust libraries interact.

C.2.1 Encountered challenges

Inclusion of `cxx_build` files in CMake The output of the `cxx_build` process is a `cc::Build` object, which is is an object called `cc::Build`, which is designed to be used with CMake (a build system for C++ projects). However, when `cxx_build` is used in conjunction with CMake, the `cc::Build` object is only treated as a "guideline" and not as an actual input to CMake. This means that the `cc::Build` object by itself does not provide enough information for CMake to properly include all the necessary files for building the C++ library. The "glue" C++ source files that are generated during the process will not be included in the build if only using the `cc::Build` object. Because of this, linkage errors will occur since the build is missing symbols, meaning that the generated C++ object files are missing in the final linking step.

To address this issue, the generated source files must be added to CMake manually. The recommended approach is to define a custom method for specifying the location of the generated files to CMake, using the `CXXBRIDGE_OUT` environment variable. This variable is set in the crate's `build.rs` file prior to calling CMake. The `CMakeLists.txt` file of the crate subsequently searches for all C/C++ source files in the specified location and adds them as sources for the library being built.

Difficulties in setting up Hunter Hunter is a CMake-based package manager for C++ that simplifies the use of external dependencies, such as `conan`. We attempted to use Hunter in our project as it is also used by `libp2p`, a library that we anticipate needing to use in the future. However, we encountered issues with the official HunterGate guide for setting up Hunter. We ultimately resolved the issue by using a local copy of Hunter. This has the added advantage that we can specify specific versions of Hunter, which is particularly useful when working with `libp2p`, since they setup their own.

Limited C++ Editor Support The process of setting up a C++ project can be quite complex, and many IDEs, LSPs, and tools have specific requirements for proper configuration and functionality. To provide proper auto-completion for each file, it is advised to provide the tools with the compilation database, so that completion can be provided and the project can be analyzed.

However, in our project the build process is initiated from the Rust side and occurs within Cargo's `target` folder, which is separate from the actual source code. As a result, even if the compilation database is exported by CMake, it will be exported in the incorrect location. To overcome this issue, we included a simple workaround in the build file that creates a symbolic link to the generated file in the correct location when the desired target is built. This results in the presence of a link to `compile_commands.json` in the `CMakeLists.txt` directory after the first build. If the file is not present, the project can be built again to generate it.

Component # Polkadot-node-core-	Substrate Dependencies		Internal Node libraries # Polkadot-	
approval-voting	sp-consensus-slots sp-application-crypto	sc-keystore sp-consensus sp-runtime	subsystem subsystem-util overseer	primitives node-primitives gum node-jaeger
av-store			subsystem subsystem-util overseer erasure-coding	primitives node-primitives gum
backing	sp-keystore		subsystem subsystem-util erasure-coding statement-table	primitives node-primitives gum
bitfield-signing	sp-keystore		subsystem subsystem-util	primitives gum
candidate-validation	sp-maybe-compressed-blob		primitives node-primitives	subsystem subsystem-util parachain
chain-api	sc-client-api sc-consensus-babe	sp-blockchain	primitives gum	subsystem subsystem-util
chain-selection			primitives node-primitives gum	subsystem subsystem-util
dispute-coordinator	sc-keystore		primitives node-primitives gum	subsystem subsystem-util
parachains-inherent	sc-network sp-blockchain	sp-inherents	primitives gum	subsystem overseer
provisioner			primitives node-primitives gum	subsystem subsystem-util
pvf-checker	sp-keystore		primitives node-primitives gum	subsystem subsystem-util overseer
pvf	sc-executor-wasmtime sc-executor-common sp-wasm-interface sp-maybe-compressed-blob	sc-executor sp-externalities sp-io sp-core sp-tracing	core-primitives node-metrics parachain	
runtime-api	sp-consensus-babe		primitives gum	subsystem subsystem-types subsystem-util

Table 3: Core submodules and their dependencies.