Pravega Client Library for .NET

Sponsored by: Dell Technologies

Icon

Description automatically generated

**By: The Pravegateers**

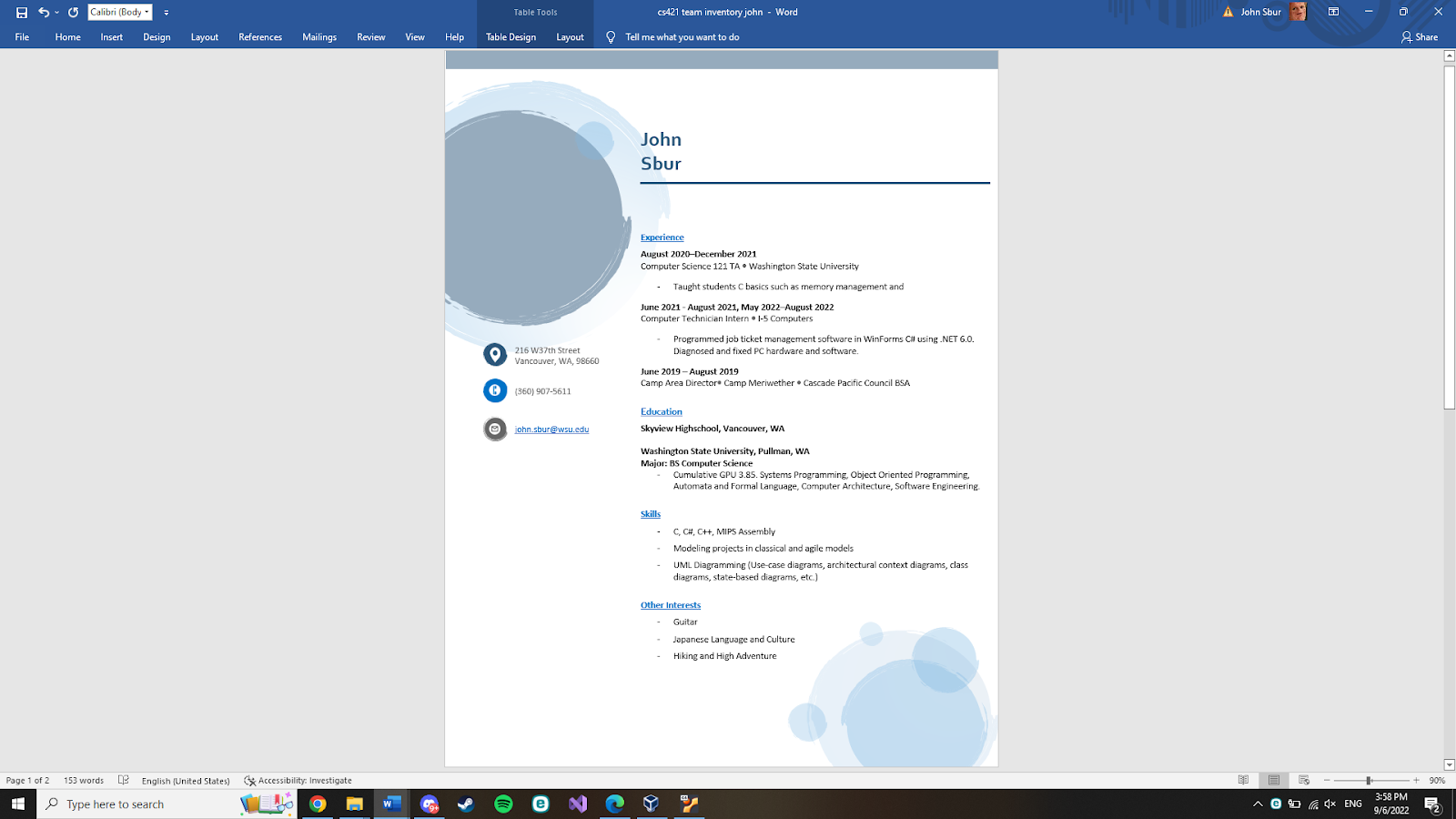
Sbur, John

Lopez, Samuel

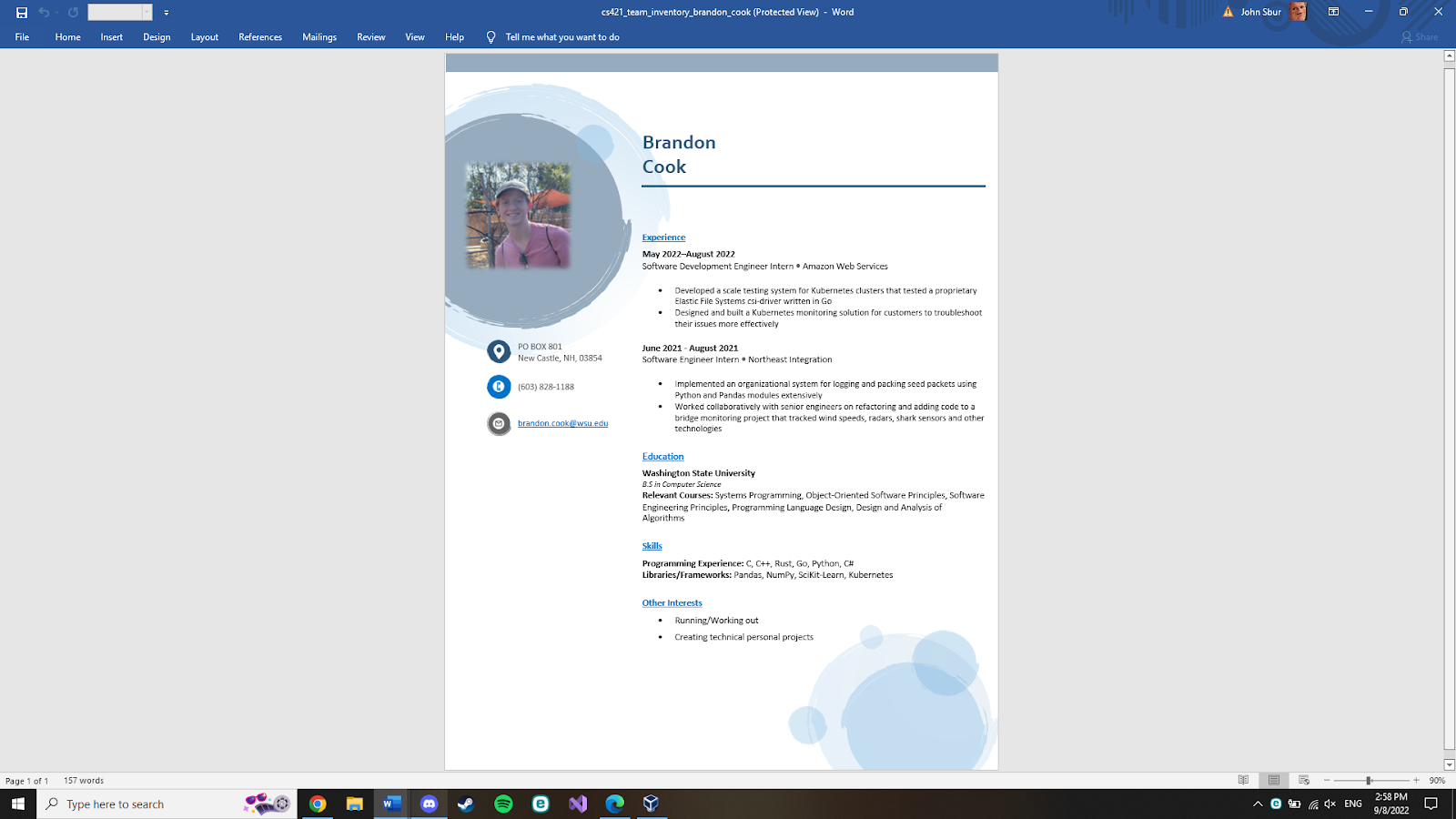
Cook, Brandon

**Table of Contents**

|  |  |
| --- | --- |
| Table of Contents | 2 |
| Team Members | 4 |
| Project Description | 7 |
| Introduction | 7 |
| Background | 7 |
| Project Overview | 7 |
| Client and Stakeholder Identification | 8 |
| Requirements and Specifications | 10 |
| Introduction | 10 |
| Project Requirement Specifications | 10 |
| Use Cases | 10 |
| Functional Requirements | 12 |
| Event Stream | 12 |
| Transactions | 12 |
| Byte Client | 12 |
| Synchronizer | 13 |
| Key Value Tables | 13 |
| Stream Manager | 13 |
| Non-Functional Requirements | 13 |
| Project Evolution | 15 |
| Solution Approach | 16 |
| Introduction | 16 |
| System Overview | 16 |
| Pros and Cons | 17 |
| Software Design | 18 |
| Library Layout | 18 |
| Library Decomposition | 19 |
| Pravega C# (Super) | 19 |
| Utility | 20 |
| Client Factory | 22 |
| Controller Client | 24 |
| Client Config | 25 |
| Byte | 27 |
| Event | 30 |
| Other Libraries | 31 |
| Data Design | 32 |
| Pravega C# (Super) | 32 |
| Utility | 32 |
| Client Factory | 33 |
| Controller Client | 33 |
| Client Config | 33 |
| Byte | 34 |
| Event | 34 |
| Shared | 34 |
| Technologies Developed and Used | 36 |
| Pravega Rust Object Representation | 36 |
| FFI | 36 |
| OnceCell | 37 |
| Callback Manager | 37 |
| PathGen | 38 |
| Asynchronous Communication Between Rust and C# | 38 |
| Test Case Specifications and Results | 40 |
| Testing Overview | 40 |
| Environment Requirements | 40 |
| Test Results | 40 |
| Project and Tools Used | 42 |
| Description of Final Prototype | 43 |
| Project Delivery Status | 45 |
| Conclusion and Future Work | 46 |
| Limitations and Recommendations | 46 |
| Future Work | 46 |
| Acknowledgements | 46 |
| Glossary | 46 |
| References | 46 |
| Appendix | 48 |
| Team Information | 48 |
| Example Testing Strategy Reporting | 48 |
| Project Management | 50 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |





Project Description

**Project Description**

1. **Introduction**

Pravega is an open-source storage system implemented and led by Dell Technologies. It uses Streams as a first-class primitive which are based on the append-only log data structure. They are flexible and have good performance [1]. By implementing clients for Pravega in multiple languages, its use can expand to a variety of applications. It currently has clients in Java, Rust, and Python.

The goal of this project is to take the existing Pravega API client that is written in Rust and to create a wrapper for it in C#. The implementation should be seamless and behave the same way as any other C# library. By doing this, the .NET Framework, one of the most popular in the world, will be able to be used with Pravega.

1. **Background and Related Work**

Dell has an open-source project called PravegaAPI which delivers a distributed storage system for elastic data streams. The Pravega client is written in Rust, however they want to be able to access it in C# for broader access. Our team needs to be able to wrap all the functionality of the PravegaAPI in C#.

Tools have already been created to help efficiently wrap the code. There is a mechanism called foreign function interfaces (FFI) that allows you to call functions in one language written in another. When using FFI with Rust, it converts the code to C. You then convert the C code into C# for this development. There is also a framework called safer\_ffi [2] which allows the Rust code to not be filled with unsafe code blocks since we will be writing to unsafe C code. Another helpful resource is a blogpost [3] talking about wrapping Rust in C#. The blogpost is especially helpful in explaining how they handle memory allocation. This is extremely important because Rust has its own ownership system and C# has a garbage collector. By just using FFI without implementing a system to handle memory, you would run into problems. Instead of manually wrapping each function in Rust, it is possible to automatically wrap them. With a tool such as interoptopus [4], it attempts to make every binding look idiomatic with a large variety of languages supported. While this tool can save time it won’t be able to wrap everything perfectly and correct.

To be able to effectively complete this project for Dell, all members on the team will have to familiarize themselves with these technologies. The first is the Rust language, each member should understand the basics, especially the barrowing system. Two great resources for learning Rust is the Rust Documentation Introduction [5] and TourOfRust [6], for a more hands-on approach. Each teammate should also research FFI, specifically for wrapping Rust. One more language all members must know is C# and .NET. Microsoft provides some interactive tutorials [7] if you need an introduction. Lastly a good understanding of the C language is important. Everyone should be able to read C code and have a clear understanding of what it is doing.

1. **Project Overview**

The main problem this project is trying to solve focuses around the Pravega API. While the API supports many languages such as NodeJS, Rust, and Python for examples, it doesn’t work in the .NET framework and C#. What this means is that developers wanting to use the Pravega API for data streaming cannot use it on their .NET applications without having to manually wrap the API themselves, which is incredibly tedious for a user. This also means that the scope of problems that Pravega can be applied to is limited as it is bound by the languages it supports and cannot support .NET applications.

The objective of this project is to create a wrapper for Pravega such that it can be interpreted by C# in a .NET framework. Doing so will allow development using Pravega by a larger pool of businesses and skilled .NET developers who can use it for critical applications. Modern businesses require efficient and secure data transportation and Pravega can help more businesses accomplish this once this project is complete. By the end of the project’s timeline, the goal is to have a wrapper or some method of processing Pravega functions such that it can be interpreted in C#. This includes all Pravega Modules, which are byte, client\_factory, error, event, index, and sync [8].

The first step in this process will be researching Rust, C#, and their similarities and differences from one another, such as how C# uses a garbage collection method for memory disposal while Rust doesn’t. Finding similarities and differences between the interfaces will allow us to plan what can be done easily between the two languages and what needs to be accounted for in development. In addition, it is important to choose a framework designed for wrapping Rust functions into C#. Testing needs to be done on an individual function basis in order to see what framework wraps Rust functions the best. After we have decided on our framework and method of converting functions from Rust and wrapping them into C#, we will begin working on each module, wrapping more basic modules before tackling more complex modules. The “byte” module is the first target.

Our major milestones in this project will be developing a method of wrapping each function that can be applied to most other functions in Rust as summarized by the previous section as well as having all modules converted from Rust to C#. By the end of the fall semester at Washington State University, roughly late November, we plan on having the first milestone at a minimum complete. This includes having researched and tested frameworks that can mediate between C# and Rust as smaller milestones completed before the wrapping method is developed. The next milestone will be pertaining to each module as each module is another step closer to the final goal. After the first module byte, we will move onto client\_factory, then index, and so on. The project ends once the final module is complete and the public Pravega API wrapper is stored onto GitHub, which our team plans on having created by the end of spring semester, roughly late April. This is to act like a C# library and to appear as such for the user. A user not familiar with Pravega should be able to use this with little issue and a transition for an experiences Pravega user should be seamless. If time allows, discussion about implementing simple cyber security principles into the library is planned, but not urgent as Rust is an extremely secure language.

1. **Client and Stakeholder Identification and Preferences**

The Client for this project is Dell Technologies. They will be guiding and assisting us with the creation of the C# wrapper. They are also our primary stakeholder, as our project will enhance their Pravega system with the ability to be used in the .NET framework, further increasing its usefulness.

Our stakeholders also include the students working on the project. By successfully completing it, not only will we fulfill the requirements to graduate, but we will gain experience working with Dell, a global technology company. Potential future users are also stakeholders in the project. They could use Pravega in future application to handle data storage and management.

The largest preference for all of our stakeholders is that the wrapper is seamlessly implemented in C#. It should be able to be called and used like any other library in C#. This will ensure its maximum usability and will increase the chances that it will be used by developers.

**Requirements and Specifications**

**I.\_Introduction**

Pravega is an open-source storage system implemented and led by Dell Technologies. It uses Streams as a first-class primitive which are based on the append-only log data structure. They are flexible and have good performance [1]. By implementing clients for Pravega in multiple languages, its use can expand to a variety of applications. It currently has clients in Java, Rust, and Python.

The goal of this project is to take the existing Pravega API client that is written in Rust and to create a wrapper for it in C#. The implementation should be seamless and behave the same way as any other C# library. By doing this, the .NET Framework, one of the most popular in the world, will be able to be used with Pravega.

At the end of the project, the C# wrapper should allow users to replicate all features found in the Rust API. It should allow end users to stream data, allowing them to read and write data in the form of bytes as well as events, eventually leading to being stored in Pravega [N].

**II. System Requirements Specifications**

**II.1 Use Cases**

Diagram

Description automatically generated

**Upload/Receive Data:**

This use case represents the overall purpose of an end user, to upload and receive their data through Pravega.

Relevant Functional Requirements: All

**Read Bytes:**

This use case represents the end user’s ability to read bytes of data from a Pravega stream.

Relevant Functional Requirements: Byte Reader

**Write Bytes:**

This use case represents the end user’s ability to write bytes of data to a Pravega stream.

Relevant Functional Requirements: Byte Writer

**Read Events:**

This use case represents the end user’s ability to read data from a Pravega Stream in the form of events. An event is a sequence of bytes.

Relevant Functional Requirements: Event Reader, Reader Group

**Write Events:**

This use case represents the end user’s ability to write data to a Pravega Stream in the form of events. An event is a sequence of bytes.

Relevant Functional Requirements: Event Writer, Reader Group

**Manage Transactions:**

This use case represents the end user’s ability to manage Pravega’s transaction feature. This allows for a series of events to be created and linked together, but the linked data can be added to the stream or abandoned as needed.

Relevant Functional Requirements: Managing Transactions

**Manage Streams:**

This use case represents the end user’s ability to manage streams. Streams are how data is written to Pravega, so giving the user control over them will allow them to maximize functionality.

Relevant Functional Requirements: Stream Manager Client

**Synchronize State**

This use case represents the end user’s ability synchronize states between processes.

Relevant Functional Requirements: State Synchronizer

**Use Key Value Tables:**

This use case represents the end user’s ability to use key value pairs in order to reduce the load that is put on the system.

Relevant Functional Requirements: Managing Transactions

**Synchronize Table:**

This use case represents the end user’s ability to update the client and server state as needed.

Relevant Functional Requirements: Table Synchronizer

**II.2. Functional Requirements**

**II.2.1. Event Stream**

**Event Writer:** The system must be able to write to the stream.

**Source:** Supported from past Pravega API wrapping projects [9]

**Priority:** Priority Level 0: Essential and required functionality

**Event Reader:** The system must be able to read from the stream.

**Source:** Supported from past Pravega API wrapping projects [9]

**Priority:** Priority Level 0: Essential and required functionality

**Reader Group:** The system must be able to collectively read all events in the stream by distributing the readers.

**Source:** Supported from past Pravega API wrapping projects [9]

**Priority:** Priority Level 0: Essential and required functionality

**II.2.2. Transactions**

**Managing Transactions:** The system must be able to manage written transactions. Managing includes committing, flushing, checking status, pinging and aborting.[11]

**Source:** Supported from past Pravega API wrapping projects [9]

**Priority:** Priority Level 0: Essential and required functionality

**Transaction Writer:** The system must be able to write events into a Transaction.

**Source:** Supported from past wrapping projects [9]

**Priority:** Priority Level 0: Essential and required functionality

**II.2.3. Byte Client**

**Byte Writer:** The client should be able to write to the stream without any headers or encoding. This design is ideal for cases such as video streaming.[10]

**Source:** Supported from some of the past Pravega API wrapping projects [9]

**Priority:** Priority Level 1: Desirable functionality

**Byte Reader:** The client should be able to read from the stream from data without headers or encoding. This design is ideal for cases such as video streaming.[10]

**Source:** Supported from some of the past Pravega API wrapping projects [9]

**Priority:** Priority Level 1: Desirable functionality

**II.2.4. Synchronizer**

**State Synchronizer:** Provides the system a way to have a state that is synchronized between many processes. [11]

**Source:** Supported from Pravega API written in Java [9]

**Priority:** Priority Level 2: Extra features or stretch goals

**Table Synchronizer:** The system compares the client state and server state and only updates if they are matching, if not it will update the client state. [12]

**Source:** Supported from Pravega API written in Rust [9]

**Priority:** Priority Level 2: Extra features or stretch goals

**II.2.5. Key Value Tables**

**Key Value Tables Client:** Uses Key-Value pairs to reduce operational burdens on the system.

**Source:** Supported from Pravega API written in Java and Rust [9]

**Priority:** Priority Level 1: Desirable functionality

**II.2.6. Stream Manager**

**Stream Manager Client:** This system is used to create Scopes, Stream, Writers and Readers.[12]

**Source:** Supported from past Pravega API wrapping projects [9]

**Priority:** Priority Level 0: Essential and required functionality

**II.3. Non-Functional Requirements**

**[Language Specification]**

The languages we use are Rust and C#. Rust was originally chosen as a language because it is low to the ground and safe as opposed to low to the ground C based languages that tend to be unsafe. C# is generally safer than C, but in this case, we program in unsafe C#, which can be quite difficult and hazardous if handled incorrectly. Both of these facts are recognized as we move forward.

**[Development Method]**

This project is developed under the design of the Agile project management method. This was chosen over the classical Waterfall project management method used in the past as it was found to be too rigid and didn’t allow for as much client input. For development, what this means is that we perform work in what are known as sprints. Sprints are periods of time where team members work on a product backlog that is based on the client’s needs, working for a couple weeks each day before reviewing and reconciling with their client before adding to a product backlog and continuing soon after. The product backlog usually is composed of features and tasks that need to be completed for the client’s product to be considered complete. With the ability to adjust to change easily with the Agile method, this method will complement our project well as problems are encountered, though it is assumed that the end goal will not change significantly.

**[Testing]**

This project gets tested each step of the way. Methods in this project are built upon each other. The basic methods need to work nearly flawlessly or future methods could experience problems leading back to basic methods. Therefore, before proceeding to more complex methods, the integrity of smaller methods need to be verified. It needs to fail when we expect it to fail and pass when we expect it to pass. A set of significant normal, boundary, and exception cases will be used during development to verify integrity. We currently us NUnit on the C# side for testing.

**[Open Source]**

This project is open-source as is specified by the client. The intent is for this project to be used by a wide range of developers as well as improved upon or used as a stepping stone into more libraries. As such, being open-source is an absolute requirement. In addition, we will be keeping a GitHub repository as our open-source code base. This also ensures we have a version history and branch capabilities for development and post-development use.

**[.NET Core]**

This project is being developed for .NET core. While there are many versions of .NET, .NET core is designed to be used by many different applications across as many compilers as possible. Therefore, since the goal of this project is to make a library that many different developers from different areas can use, it makes sense to use .NET core.

**[Dependence]**

This system does not depend on Operating System specific libraries. This is for the same reason as we are using .NET Core as using Operating System specific libraries would mean that this project can only be used on those Operating Systems. The system shouldn’t depend on these so that it can apply to different Operating Systems using .NET Core. In addition, this library we are creating is be self-contained and not instantiate other external processes for the same reasons. We want this to be as generic as possible and not have to be coupled with other processes and libraries for it to function.

**[Style]**

This system uses C# naming conventions. The idea of this project is for this library to appear and be used like any other C# library. Standard naming conventions will be used in the naming and writing of code so that a developer in C# doesn’t have to worry about having to treat this library differently from other libraries they are using.

**[Documentation]**

This project is documented each step of the way. Each coding and non-coding process in the project needs to be documented, not just so that an outside user can understand it, but also so that our process makes sense to our client. Furthermore, documenting each step and function in detail will make it easier to build upon the project in the future. We will be using a program similar to StyleCop, a C# program designed to enforce naming and documentation conventions, to ensure that our documentation is consistent and thorough.

**[The System]**

This system is be seamless for C# users. The use of this library is similar to any other C# library. For people not familiar with Pravega, it appear as though they are using a generic data streaming library and not have to worry about the complexity of Pravega. This project is designed to make Pravega accessible to all C# .NET framework users so the transition from other library to this one should be as easy as possible.

This system also handles computer memory in a way that works with both Rust’s memory management system and C#’s garbage collection system. Both languages handle memory much differently than each other. C# handles memory by dumping no longer used objects into a garbage collector to be emptied later while Rust handles memory by transferring ownership of memory between processes until it isn’t transferred and the memory is destroyed, i.e., the process that owns that memory ends.

This system logs errors through Rust and sends them through C#. The idea behind this is so that a programmer that encounters an error while using the library can still find errors that were caused through the Rust library in C# instead of generic exceptions being thrown.

Finally, this system upholds all Pravega’s features. The idea is that this library created from this project is used because of the Pravega features provided. For example, one of the main features is being able to handle many data writers and readers without compromising time. If this library compromises time, then there is less reason to use this library over other data streaming libraries already available in the .NET framework. This is a near-perfect mirror of what Pravega provides in Rust already. Our goal is for this wrapper to operate time-wise at an average of 85% efficiency at least, meaning that the functions wrapped in C# are at least 85% as fast as the Rust functions. Time is expected to be lost while wrapping, but the goal is to lose as little time as possible.

**III. Project Evolution**

As this project is being planned, the underlying assumption is that the project vision will not change, and the end goal will remain the same throughout development. However, if the initial plan alters through development, the Agile project management method implemented should prepare the team well. Any unintended changes can be added to the project backlog as features to be implemented with varying levels of priority. The downside to this method of work is that if too many features and changes are requested, it could mean drastic system overhauls that will take a significant amount of time to implement. Therefore, methods are kept simple and applicable to many situations, and documentation is written throughout the library to minimize the potential impact. Some potential features that can be implemented if requested are security measures for encrypted data transfer, as well as using LINQ within the C# framework. These are mentioned as features to add if time allows it, especially LINQ because of its powerful capabilities.

**Solution Approach**

**I.\_Introduction**

Pravega is an open-source storage system implemented and led by Dell Technologies. It uses Streams as a first-class primitive which are based on the append-only log data structure. They are flexible and have good performance [1]. By implementing clients for Pravega in multiple languages, its use can expand to a variety of applications. It currently has clients in Java, Rust, and Python.

The goal of this project was to take the existing Pravega API client that is written in Rust and to create a wrapper for it in C#. The implementation is seamless to the user and behave the same way as any other C# library. With, the .NET Framework, one of the most popular in the world, users can now start using the Pravega technology in any .NET supported languages.

**II. System Overview**

This system is designed to transition Rust code into executable C# code that works as nearly as well as the Rust API. Functionally, it respects both language’s limitations and properties while keeping functionality. A user can view the project as an input output system where it takes in Rust code and outputs C# code. As of now the modules that are implemented are client factory byte reader, byte writer and event wrtiter. Each of these modules had their own complex data structures taken into consideration while wrapping as these data structures.

The way this project is designed, is to convert the modules and their content through the wrapper is by using an FFI, a mechanism that lets one programming language use code from another, to generate the code in C and then generate it into C#. Initially, we used Interoptopus to help with creating an FFI architecture, but later in development it became easier to generate our own. The advantage of using a wrapper is that it’s converting the code to C# with restrictions we put in place. Any changes made to the Rust code should not have a large impact on the C# code as the wrapper with said restrictions will account for that. The wrapper model will be less coupled as a result. A benefit to this solution is any bug fixes on the Rust side will also be patched on the C# side. Each public function is specifically programmed to convert the data needed from the C# side to Rust and back. Since both Rust and C# recognize C, we can go from C to C# completing the transition.

When designing our solution, several factors were considered. An important aspect is that Pravega modules are coupled with other modules that relates to their functions. As mentioned earlier, Rust data structures need to convert well to C# to ensure the continuous support of Pravega functionality. In addition, to limit coupling of this system to specific operating systems and applications, the system is mostly self-contained. This means that it doesn’t use any external processes and does not use any operating system specific libraries. The system is coupled to .NET core, the system the project is built upon, but since .NET core can be ran on most operating systems, this shouldn’t conflict with the goal to have the system be accessible to a large audience a significant amount.

One big part of Rust that needs to be integrated is the synchronous functionality that Rust provides. Rust and C# handle synchronous tasks differently between each other. Because of the large differences the current solution calls the asynchronous functions but are not run asynchronously. With the way our solution is designed, once there is a way found to run the Pravega code asynchronously, it will be easy to implement.

**III. Pros and Cons**

**Pros**

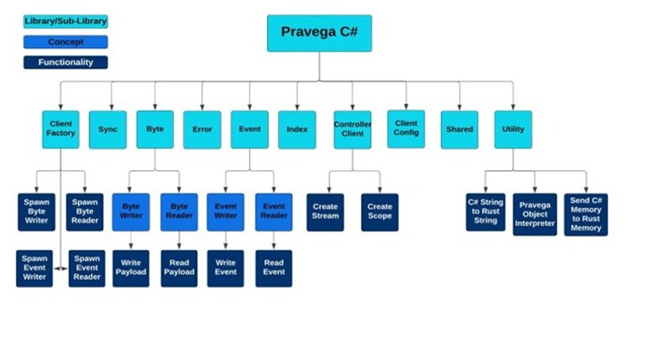
* Updating the Rust code will automatically update the corresponding C# code, streamlining the development process.
* The implementation process is faster, as there is fewer design patterns need to be found compared to a complete implementation on C#
* Design is able to reflect the original Rust design, so moving between C# and Rust development is seamless

**Cons**

* There are limited resources available for communication between Rust and C# languages, as it is a relatively unexplored area.
* Communication between languages incurs additional time and memory costs that can potentially impact performance.

**Software Design**

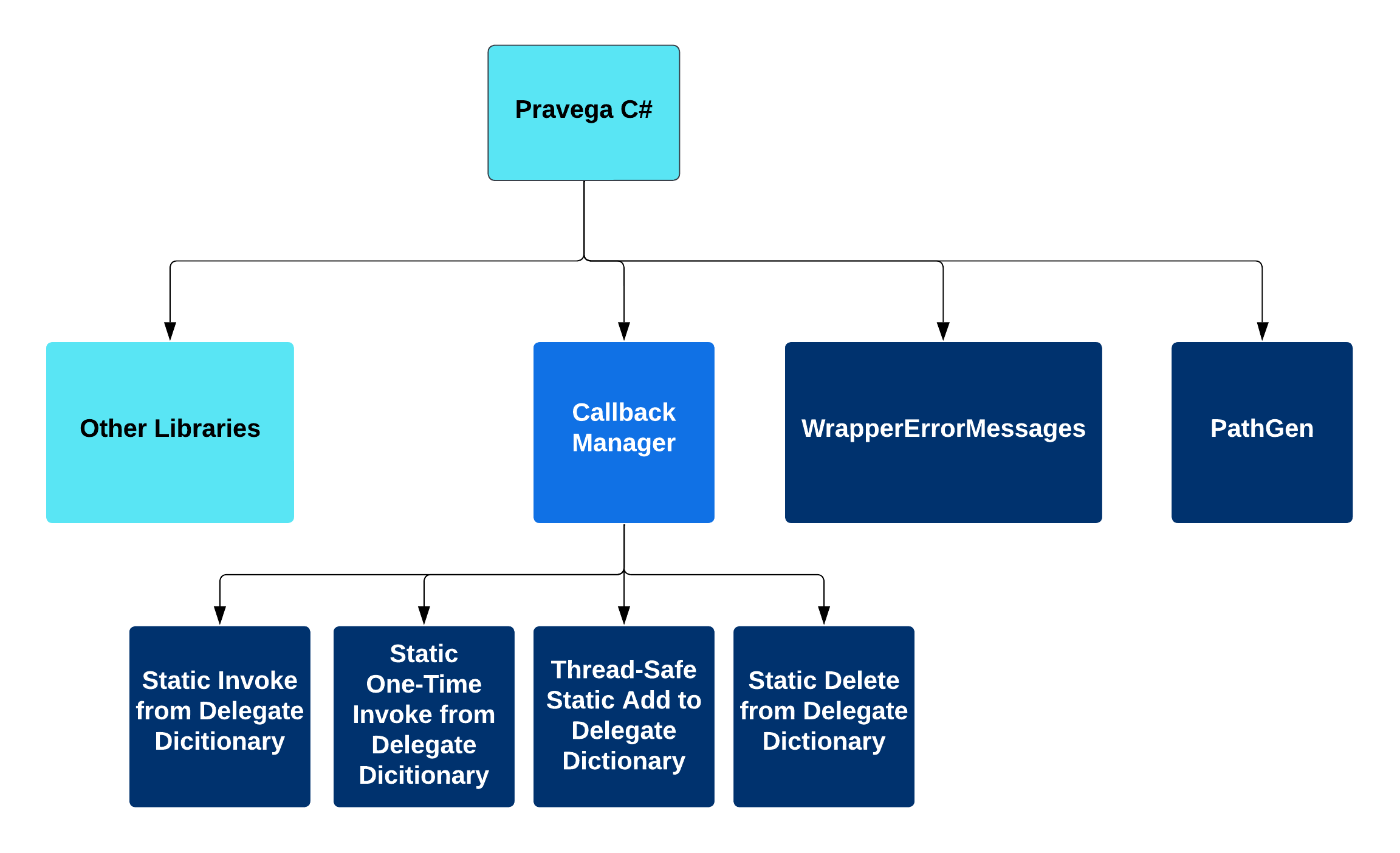
**I. Library Layout**



Our system is represented by the C# Wrapper and acts like a library. Its super systems include the Pravega library, its peers include the Pravega Servers, its clients include C# Developers, and its subsystems manifest as the modules within the wrapper, those being Byte, Event, Index, Client Factory, Error, and Sync. In addition, more libraries were implemented based on smaller features both Pravega and FFI related, those being Controller Client, Client Config, Shared, and Utility. This wrapper comes directly from the Pravega library as a sub system meant for C# development making the Pravega rust dll its parent. The dll provides the Rust code wrapped into C# code. It communicates with Pravega Servers for data streaming and Pravega servers send information back to the wrapper client making them peers. The subsystems are the components that make up the wrapper. They are the modules of the Pravega Rust library manifested through the wrapper and there is some dependence on other subsystems between each subsystem. All of these components make up the core of our project.

**II. Library Decomposition**

**[Pravega C# (Super)]**



This is the core of the library and so methods and objects here are useful to all sub libraries underneath it. Similar to the Utility library.

**Services Provided:**

**Service**: Callback Manager

-Provided to: Library developer

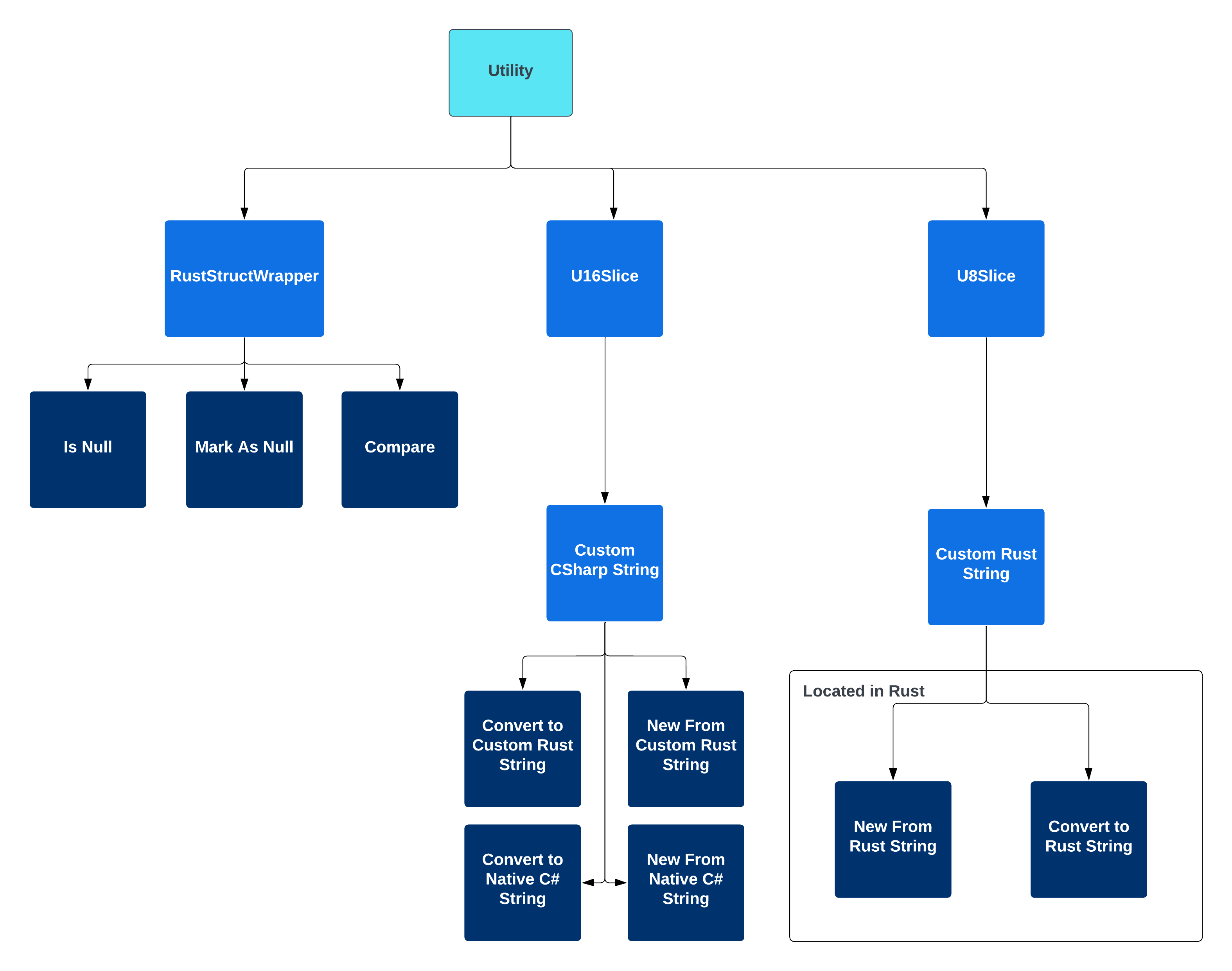
-Description: A technology developed for allowing safe communication between unmanaged languages and managed languages. This manager holds delegates, or temporary anonymous methods. Then, it allows Rust, operating in unmanaged memory, to call specific static methods in C#, which are in managed memory but do not move in memory. The methods Rust calls then allow the calling of delegates from Rust by redirecting Rust to them in memory which, though they move around, is safe since C# knows where managed memory objects and methods are.

**Service: Library error messages**

-Provided to: Library developer and C# developer

-Description: Provides descriptive error messages of errors related to the library that may occur on the C# side.

**[Utility]**



Much of our project focused around researching and developing methods to communicate between C# and Rust through FFI testing and other methods. That research culminates in this library. Many of the services it provides are fundamental for sending data between languages, managing memory between languages, and managing Rust objects in C#.

**Services Provided:**

**Service: CustomCSharpString**

-Provided to: C# Developer

-Description: Allows for the creation of strings in C# in unmanaged memory. A standard string in C# usually falls into managed memory and is able to be collected by the garbage collector so for use in Rust the string has to be in unmanaged memory. Therefore, this string allows for the sending of strings between languages, from C# to Rust. C# strings are notably comprised of 16bit characters.

**Service: C# to Rust String conversion**

-Provided to: Library Developer

-Description: Because strings in C# are 16bit and strings in Rust are 8bit, a conversion between the two is necessary when dealing with sending strings from C# to Rust. The class CustomCSharpString and CustomRustString pprovides methods for converting between 16bit strings into 8bit strings in both C# and Rust with data loss being necessary to represent the Rust string from C#. Split into three steps where the C# string is converted to an unmanaged 16bit array, that array is converted into an unmanaged 8bit array, and then finally that array is converted into a native Rust string. Steps can occur individually as per the developer’s needs.

**Service: Rust to C# String conversion**

-Provided to: Library Developer

-Description: Because strings in C# are 16bit and strings in Rust are 8bit, a conversion between the two is necessary when dealing with sending strings from Rust to C#. The class CustomCSharpString and CustomRustString provides methods for converting between 8bit strings into 16bit strings on both the Rust and C# side with no data loss. This operation occurs in unmanaged memory. Split into three steps where the Rust string is converted into an 8bit array, that array is converted into a 16bit array, and then finally that array is converted into a C# native string. Steps can occur individually as per the developer’s needs.

**Service: Unmanaged C# string to managed C# string**

-Provided to: C# developer

-Description: Transforms a string located in unmanaged memory into a string in C# managed memory that is able to be garbage collected.

**Service: Rust struct wrapper**

-Provided to: Library developer

-Description: Provides a way for C# to keep track of Rust objects held in unmanaged memory. Serves as the super class to be inherited from for complex Pravega objects and acts as a way to communicate with memory in Rust, allowing Rust to do lots of heavy lifting.

**Service: Rust struct compare**

-Provided to: Library developer and C# developer

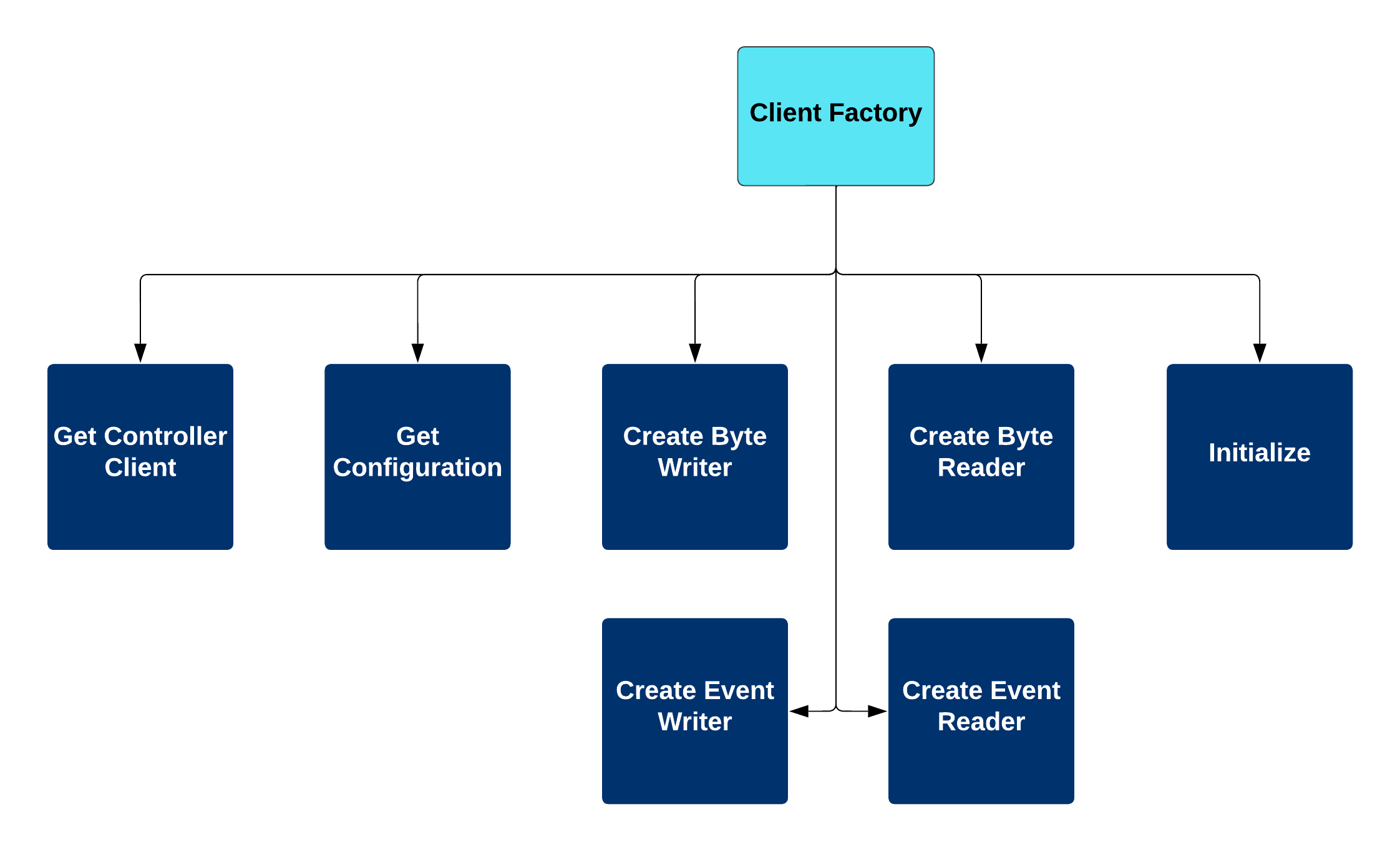
-Description: Allows for the comparison of two objects in Rust in C#. Tells the developer whether the objects are the same or not

**Service: Mark Rust struct as null**

-Provided to: Library developer and C# developer

-Description: Allows C# to “forget” where the Rust object it is keeping track of is by setting the pointer to null. Does not deallocate the objects from memory as that is the responsibility of other FFI methods.

**[Client Factory]**



This library is responsible for the creation of many integral Pravega components needed for operation. It also manages the asynchronous runtime used in Rust in the background. In addition, this library helps communicate with Rust code from C# with the help of FFI formatted functions specifically for Client Factory related services and testing.

**Services Provided:**

**Service: ClientFactory**

-Provided to: C# Developer

-Description: This service revolves around the Client Factory object in the library, a static object with the ability to be instantiated with different settings. There is only one Client Factory object that keeps track of the runtime and produces Pravega objects and this is true on the C# and Rust side.

**Service: Client Factory Initialization**

-Provided to: C# Developer

-Description: Allows a developer to initialize the library’s one client factory with different customizations. Developers can specify behavior through giving the initializer a client configuration and a specific Rust runtime can be used if given to the initializer.

**Service: Create Byte Reader**

-Provided to: C# Developer

-Description: Allows for the creation of byte reader. The produced object’s behavior is managed in the Byte module.

**Service: Create Byte Writer**

-Provided to: C# Developer

-Description: Allows for the creation of byte writer. The produced object’s behavior is managed in the Byte library.

**Service: Create Event Reader**

-Provided to: C# Developer

-Description: Allows for the creation of event reader. The produced object’s behavior is managed in the Event library

**Service: Create Event Writer**

-Provided to: C# Developer

-Description: Allows for the creation of event writer. The produced object’s behavior is managed in the Event library

**Service: Get Controller Client**

-Provided to: C# Developer

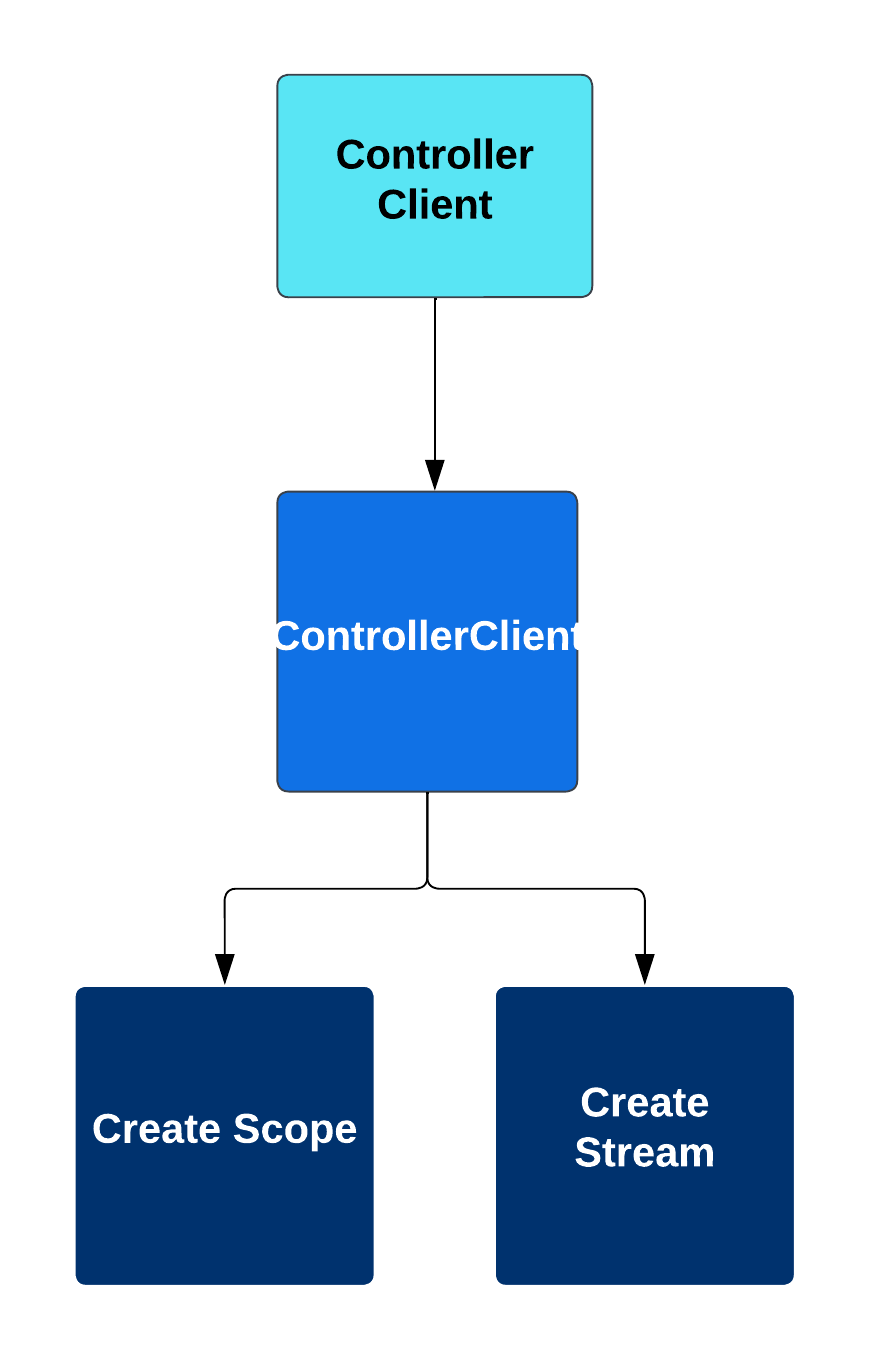
-Description: Allows a developer to get client factory’s controller which helps with the creation and modification of streams and scopes on the client factory’s Pravega Server.

**Service: Get Client Configuration**

-Provided to: C# Developer

-Description: Allows a developer to check the configuration settings of the client factory.

**[Controller Client]**



Provides a way for C# to create and manage scopes and streams on a Pravega server. In addition, this library helps communicate with Rust code from C# with the help of FFI formatted functions specifically for ControllerClient related services and testing.

**Services Provided:**

**Service: Create Scope**

-Provided to: C# Developer

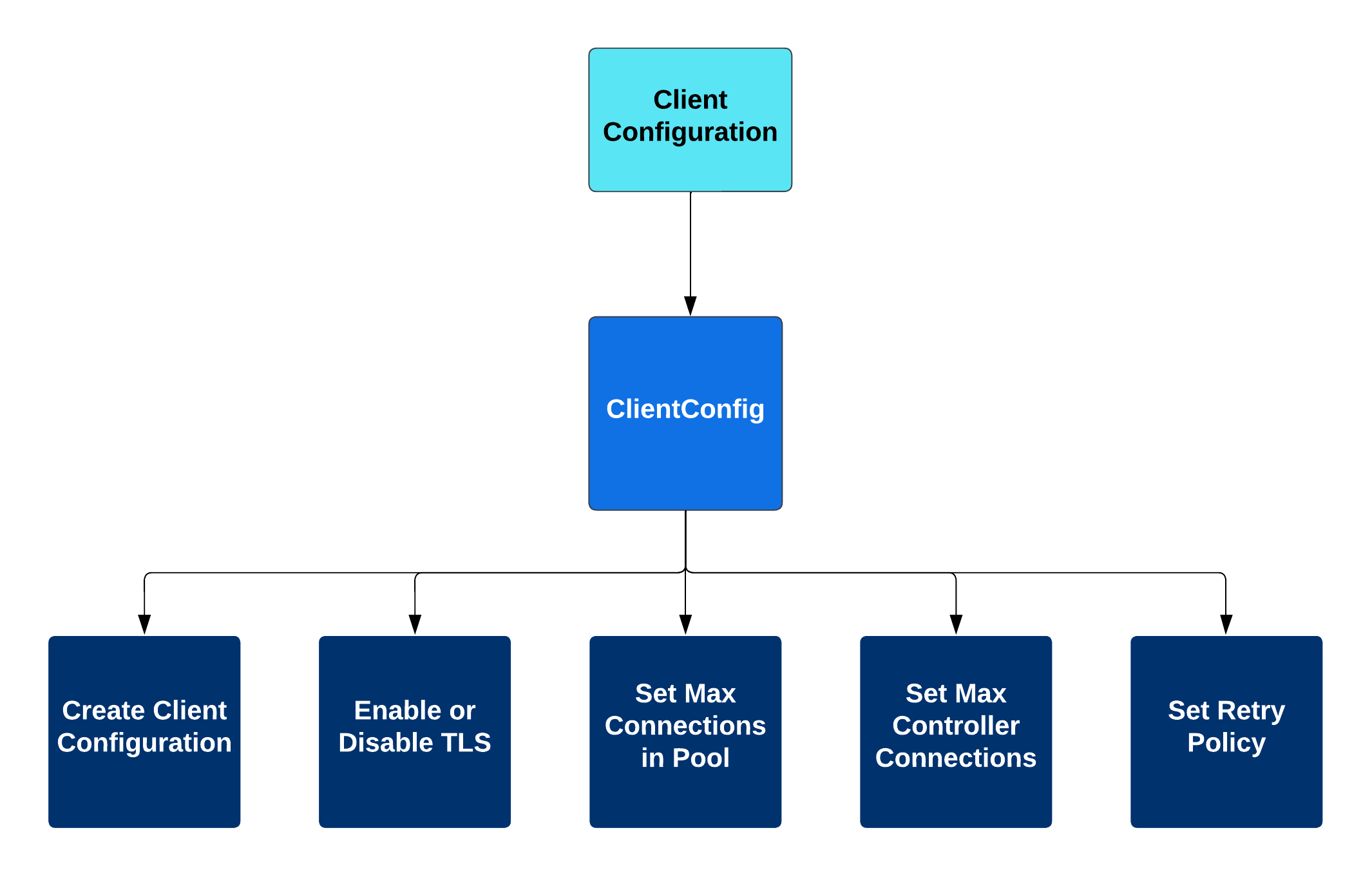
-Description: Allows the creation of scopes, environments to hold streams in, on a Pravega server.

**Service: Create Stream**

-Provided to: C# Developer

-Description: Allows the creation of streams based on scopes on a Pravega server.

**[Client Config]**

****

Provides objects and methods for creating objects that represent a user’s setting configuration. Configurations can then be applied to the Client Factory for dictating interactions with the Pravega server. In addition, this library helps communicate with Rust code from C# with the help of FFI formatted functions specifically for Client Configuration related services.

**Service: Set Max Connections in Pool**

-Provided to: C# Developer

-Description: Allows adjustment of how many connections the client factory can support on the Pravega server.

**Service: Set Max Controller Connections**

-Provided to: C# Developer

-Description: Allows adjustment of how many controller clients are allowed connected to the server.

**Service: Set Retry Policy**

-Provided to: C# Developer

-Description: Retry policy corresponds to how the server should act when a connection fails. How many times should it try to reconnect or how many reconnections it should try can be set through the retry policy.

**Service: Set Transaction Timeout Time**

-Provided to: C# Developer

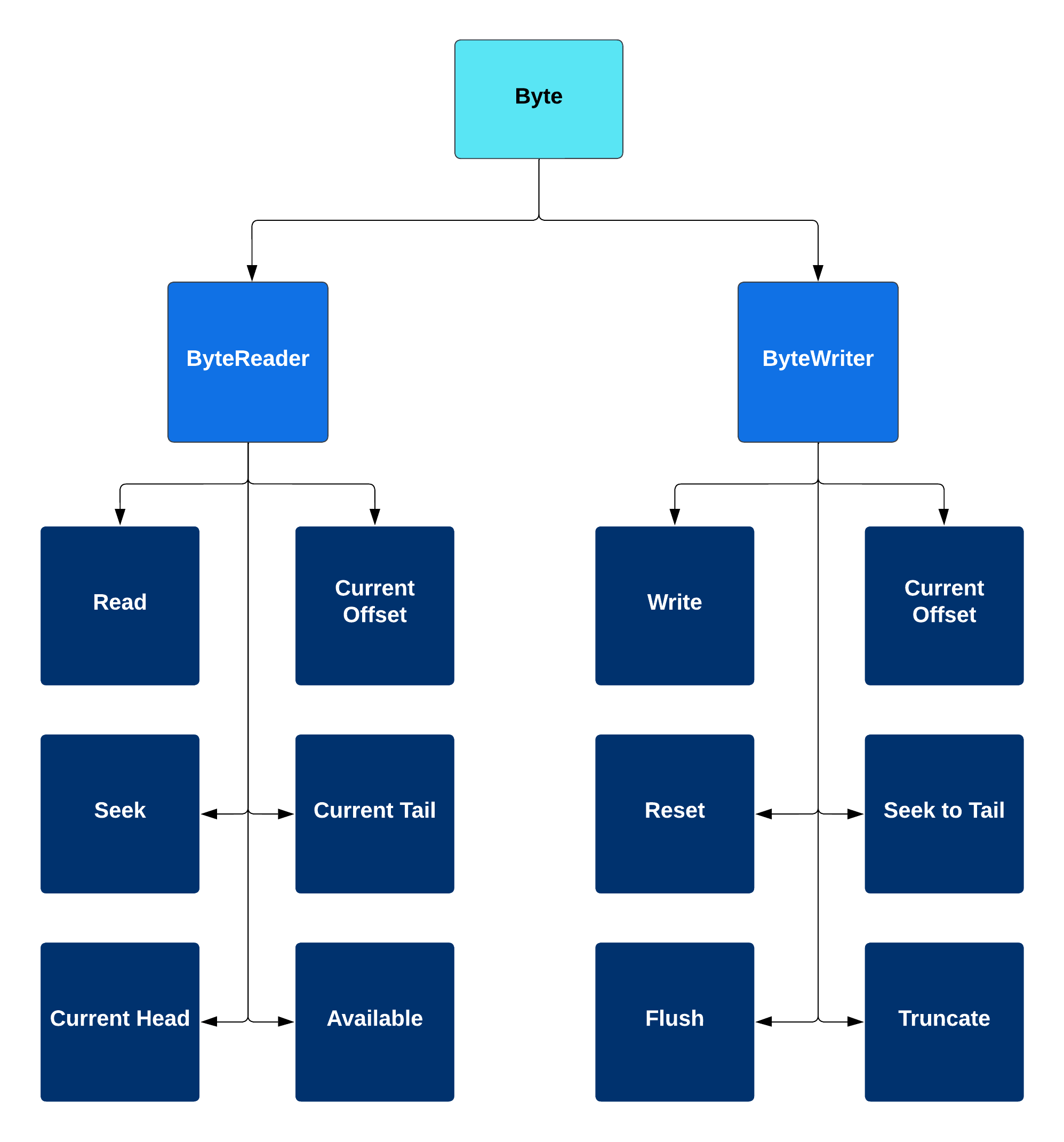
-Description: Allows the setting of how much time needs to pass for a server transaction to be considered a failure and timeout.

**Service: Enable or Disable TLS**

-Provided to: C# Developer

-Description: Allows enabling or disabling of server tls connection.

**[Byte]**

****

Allows for reading and writing of byte payloads to and from a Pravega server’s streams. In addition, this library helps communicate with Rust code from C# with the help of FFI formatted functions specifically for Byte related services and testing.

**Reader**

This subsystem is responsible for reading byte payloads correctly from a stream. It has an algorithm that prefetches data from the server and caches it into memory. Byte Readers are created from a scope and a stream within that scope. Allows the user to read from the stream safely.

**Services Provided:**

**Service: Read**

-Provided to: C# Developer

-Description: Given a number of requested bytes, the read function reads as many available bytes from its stream from its current position on that stream and returns it to the developer as a byte payload. The number of bytes returned may be less than the amount requested and this is normal behavior.

**Service: Check Availability**

-Provided to: C# Developer

-Description: Checks the availability of a reader, whether it is in use asynchronously or not.

**Service: Check Current Offset**

-Provided to: C# Developer

-Description: Checks a reader’s current position on the stream and relays it to the user.

**Service: Seek**

-Provided to: C# Developer

-Description: Allows a reader to seek along a stream to a position, either from the beginning of the stream, its current position, or the end of the stream.

**Service: Check Current Head**

-Provided to: C# Developer

-Description: Looks for the safest currently readable head, or the point on the stream closest to the 0 position that can be read, and relays the information to the user.

**Service: Check Current Tail**

-Provided to: C# Developer

-Description: Looks for the safest currently readable tail, or the point on the stream closest to the end of the stream that can be read, and relays the information to the user.

**Writer**

This subsystem is responsible for writing the byte payloads correctly. The system will only let you write 8 MiB at a time. Byte Writers are created from a scope and a stream within that scope. Allows the user to write to the stream safely.

**Services Provided:**

**Service: Write**

-Provided to: C# Developer

-Description: Given a byte payload, a Byte Writer is able to write that payload of bytes to its stream.

**Service: Check Current Offset**

-Provided to: C# Developer

-Description: Checks a writer’s current position on the stream and relays it to the user.

**Service: Flush Stream**

-Provided to: C# Developer

-Description: Allows a byte writer to flush all data from a stream.

**Service: Truncate Stream**

-Provided to: C# Developer

-Description: Allows the truncation of a stream. Given an offset, this service allows the deleting of all prior bytes up until that offset.

**Service: Seal Stream**

-Provided to: C# Developer

-Description: Allows the sealing of a stream, preventing further writing of payloads to that stream from any writer.

**Service: Reset Stream**

-Provided to: C# Developer

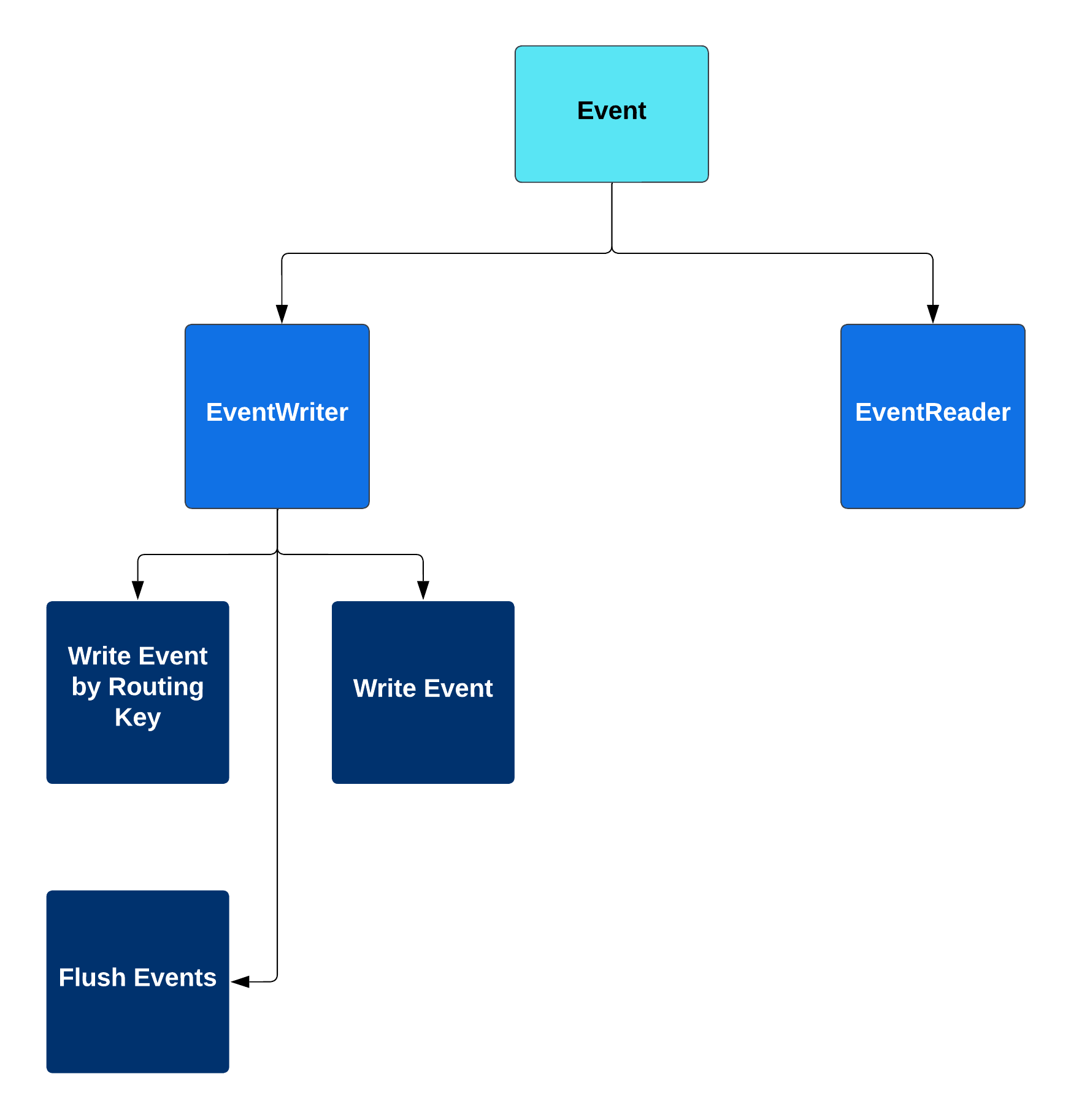
-Description: Resets the server’s internal reactor, removing incomplete actions.

**Service: Seek to Tail**

-Provided to: C# Developer

-Description: Sets the position of the writer to the end of the stream, making sure future payload writing doesn’t overwrite existing data.

**[Event]**

****

The event subsystem is responsible for reading and writing events to Pravega. It also holds the capabilities for using transactions, in which the data for an event is written, but the user can decide whether or not to commit the event to Pravega.

The subsystem was broken down based on the structs used in the rust code for the Pravega client. The structs were reimagined as C# classes whose interfaces match how the structs are connected in rust. For example, in the rust code the EventReader struct uses the ReaderState struct. So, in correspondence the EventReader Class uses the ReadeState as a variable.

**EventReader**

Currently unimplemented, though event is still acceptably done. Responsible for reading for reading events from the server.

**EventWriter**

The EventWriter is used to write events to the end of a stream. It is created by a clientfactory and uses a stream in its construction.

**Service: Write Event**

-Provided to: C# Developer

-Description: Allows the writing events to the end of a Pravega stream, generating a routing key when doing so.

**Service: Write Event by Routing Key**

-Provided to: C# Developer

-Description: Allows the writing events based on a generated routing key to a Pravega Stream.

**Service: Flush Events**

-Provided to: C# Developer

-Description: Allows deletion of events that have already completed.

**[Other Libraries]**

Other libraries listed, those being shared, index, sync, and error, do not provided any explicit service, but either serve to hold organized objects used in other libraries or the foundation of future work. Eventually, index, sync, and error will provide their own services in future work for Pravega streaming, but for now are just placeholders.

**IV. Data Design**

**[Pravega C# (super)]**

**PravegaCSharp.cs**

Static Class Interop

-A shared static class across all libraries that use FFI communication with Rust. Provides paths to Rust generated libraries, interop methods in different libraries, and the CallbackManager for helping Rust safely communicate with C#.

**[Utility]**

**Utility.cs**

Class RustStructWrapper

-Provides a super class used for representing objects in unmanaged memory created in Rust.

Class TokioRuntime

-Represents a Rust asynchronous runtime. Can be created or destroyed from C#.

Class TokioHandle

-Represents a Rust asynchronous runtime’s handle. No methods required for implementation.

Struct U8Slice

-Represents an array of bytes in unmanaged memory. Able to be parsed, read from, and written to. This can be used in Rust safely over FFI.

Struct U16Slice

-Represents an array of 16bit objects in unmanaged memory. Able to be parsed, read from, and written to. This can be used in Rust safely over FFI.

CustomCSharpString

-Represents a 16bit string in unmanaged memory. Provides the ability to transform CustomRustStrings into CustomCSharpStrings, CustomCSharpStrings into CustomRustStrings, CustomCSharpString into native C# strings, and native C# strings into CustomCSharpStrings.

CustomRustString

-Represents an 8bit string in unmanaged memory. No methods located in C#. This can be used in Rust safely over FFI.

**lib.rs (under UtilityWrapper)**

Struct U8Slice

-Represents an array of bytes in unmanaged memory. Able to be parsed, read from, and written to. This can be used in C# safely over FFI.

Struct U16Slice

-Represents an array of 16bit objects in unmanaged memory. Able to be parsed, read from, and written to. This can be used in C# safely over FFI.

CustomRustString

-Represents an 8bit string in unmanaged memory. Provides the ability to transform native Rust strings into CustomRustStrings and CustomRustStrings into native Rust strings. This can be used in C# safely over FFI.

**[Client Factory]**

**ClientFactory.cs**

Static Class ClientFactory:

* Provides all services in the Client Factory section of Library decomposition, allowing for the creation of many fundamental Pravega objects as well as managing the asynchronous runtime. Methods implemented include Initialize (default, with ClientConfig, or with ClientConfig and Runtime), Runtime, Handle, FactoryControllerClient, Config, ToAsync, CreateByteWriter, CreateByteReader, CreateEventWriter, and CreateEventReader. All methods use FFI with Rust.

**[Controller Client]**

**ControllerClient.cs**

Class ControllerClient

-Provides the ability to create scopes and streams. Requires a ClientFactory to be created. Methods implemented include CreateScope and CreateStream. Stream creation requires a Scope and StreamConfiguration. Inherits from RustStructWrapper. All methods use FFI with Rust.

**[Client Config]**

**Config.cs**

Class ClientConfig

-Provides methods for making settings used to apply to Client Factory. Inherits from RustStructWrapper. Methods implemented include MaxConnectionsInPool, MaxControllerConnections, RetryPolicy, ControllerUri, TransactionTimeoutTime, Mock, IsTlsEnabled. All methods use FFI with Rust.

**[Byte]**

**Byte.cs**

Class ByteReader:

* This object allows the system to read raw bytes from a stream. Provides all services listed in library decomposition under the Byte library in the Reader section. Inherits from RustStructWrapper. The public methods implemented for the ByteReader class are, new, CurrentOffset, Available, Seek, Read, CurrentHead, and CurrentTail. Internal methods include InitializeByteReader and GenerateByteReaderHelper, which use Client Factory to help with the creation of a ByteReader. All public methods as well as GenerateByteReaderHelper use FFI with Rust.

Class ByteWriter:

* This object allows the system to write raw bytes to a stream. Provides all services listed in library decomposition under the Byte library in the Writer section. Inherits from RustStructWrapper. The public methods implemented for ByteWriter struct are new, CurrentOffset, Write, Flush, Seal, TruncateDataBefore, SeekToTail, Reset. Internal methods include InitializeByteWriter and GenerateByteWriterHelper, which use Client Factory to help with the creation of a ByteWriter. All public methods and GenerateByteWriterHelper use FFI with Rust.

**[Event]**

**Event.cs**

Class EventReader:

* This struct is used to read events from a segment. One thing to note is that the EventReader must belong to a reader group. It holds a pointer to a rust EventReader struct. Currently can be generated, but holds no methods.

Class EventWriter:

* This is the class used to write events. It has the function WriteEventByRoutingKey..

**[Shared]**

**Shared.cs**

Class Scope

* Inherits from CustomCSharpString. Represents the name of an environment that can contain Streams on a Pravega server.

Class Stream

* Inherits from CustomCSharpString. Represents a Pravega Stream’s name.

Class Segment

* A class representing a specific Pravega Stream Segment through its unique id.

Class ScopedStream

* Contains a Scope and a Stream. Represents a path to a Stream within a Scope.

Class ScopedSegment

* Contains a Scope, Stream, and Segment. Represents the path to a Segment on a Pravega Stream within a Scope.

Enum ScalingType

* Represents how a stream scales when it nears capacity and needs to increase in size. 0 means a fixed number of segments, 1 means scaling in kilobytes per second, and 2 means scaling in events per second.

Class Scaling

* Uses ScalingType, a TargetRate, and a ScaleFactor to dictate scaling behavior. Used to apply to a StreamConfiguration

Enum RetentionType

* Represents how a stream’s retention is determined. 0 means no retention type, 1 means by time, and 2 means by size.

Class Retention

* The Retention specifies whether truncation happens based on size or time in a Pravega Stream. It contains a minimum limit (min limit) and a maximum limit (max limit) such that the Controller truncates data respecting those limits. For example, a size-based policy specifies the amount of data to retain, so if a retention policy specifies a minimum limit = 20 GB and a maximum limit = 100 GB, then the truncation cycle ensures that the Stream has at least 20 GB and not more than 100 GB.

Class StreamConfiguration

* Much like ClientConfig for Client Factory, though StreamConfiguration focuses on the behavior of a specific stream within a scope and can be applied to it as such. Contains a Retention, Scaling, ScopedStream, and list of tags that dictate the creation and behavior of a stream. ScopedStream represents the stream to be created or the stream it is applied to.

Other Classes and objects represented here act as placeholders for future work on the library as they will be needed for development of future Pravega objects and are not required for immediate functionalities required. Those are PravegaNodeUri, DelegationTokenProvider, SegmentMetadataClient, RawClientImpl, AsyncSegmentReaderImpl, and TxId.

**V.­\_Technologies Developed and Used**

Throughout this project, our team had to find unique solutions to this problem as this area in Computer Science. Communication between managed languages and unmanaged languages isn’t unheard of, but since Rust is a new language at the time of writing this, not many projects have been done in this field. Few resources existed online and therefore we had to design our own solutions from scratch or use existing ideas.

**V.1 Pravega Rust Object Representation**

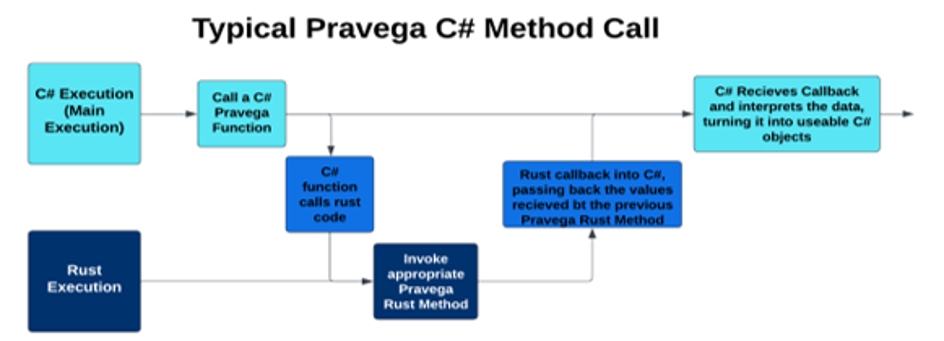
Our team went through many designs when asking the question of how to convert Rust objects into C# or how to represents Rust objects in C#. One of the first solutions was having all data in C# representing some form of the Rust data and then when we wanted to communicate between the two languages, we would pass all the data into an FFI function and build the Rust object in Rust from the inputted data. This was found to be impossible since some Rust objects cannot be represented in C# effectively. C# has no idea what Traits are, which are similar to C#’s interfaces. It then dawned on us that C# didn’t need to know what a Trait was nor how to use it. All the heavy lifting could be done on the Rust side if C# had the bare minimum of information needed to use an object and that is exactly what our solution does.

The way this architecture works is by first calling a function into Rust. An object in Rust is created and stored in unmanaged memory through a Rust Box. Boxing an object stores it in unmanaged memory and prevents it from being deleted unless the box is deleted. Then, a pointer to that box is given back to C# to store in an object. That C# object becomes the representation of the Rust object. When C# wants to call a function the Rust object has, it passes the pointer it has into Rust to which that pointer is utilized to call the desired function and the output is given back to C#. This works quite well in practice and was adopted into many of our objects in this library.

One downside of this method is that it indirectly inherits the properties of Rust object, those being object consumption. In Rust, unlike most languages, if a block of code has ownership of an object and that block completes with the object in it, that object is “consumed” or deleted. This normally isn’t an issue in most cases as we pass in the pointer to an object from C# into Rust and since pointers aren’t the actual object, the object stored in the box isn’t consumed. This isn’t the case in other objects though and sometimes in our function calls objects are consumed. It is easy to see when something will be consumed in Rust and so default behavior is that if an object is consumed in Rust, we set the fields of that object’s representation in C# to null along with its pointer.

**V.2 FFI**

FFI is not a new concept, but it is in the context of Rust and C# communication. The fundamentals have been covered in a previous section, however as an overview, the communication works by finding a bridge between the two languages. Both Rust and C# are based on C and therefore if we write code that looks like C code, that code can be understood by both languages. For our project, a lot of objects had to be dulled down into C compatible code and funneled through that into the other language. As a typical use case scenario, here is a diagram representing the process.



Following from the first part, C# initiates a call to a function that uses FFI, taking C# objects as inputs. Then, in C#, the objects are converted into C compatible objects which are then put into a DLL call into Rust code. The DLL call looks for a C formatted function. On the Rust side, the corresponding method is made to look like a C function and takes the inputs from C#. Then, the Rust code converts the inputs into whichever objects it needs before calling the appropriate method. The output of that method is retrieved and sent back to C#. In some cases, outputs are first boxed and then its pointer is sent back to C# if the object is too complex to be represented in C#. Finally, C# retrieves the output and puts it into a form it can better use before proceeding with normal execution. This is the fundamental FFI architecture we use in our library.

**V.3 OnceCell**

OnceCell is a library we utilize in Rust for safety in Client Factory. Our library only needs one Client Factory and in C# there isn’t an issue representing Client Factory as a static class. However, because Client Factory is complex in its own right, it has to use the Pravega Rust Object Representation architecture. This poses some problems as if Client Factory is deleted or tampered with during asynchronous operations, it would be disastrous for the library’s integrity. OnceCell provides an easy solution to this problem. A OnceCell object can hold one object of any type and once inside can be seen by different asynchronous threads, but never deleted. Client Factory in our library is stored in a OnceCell object for this reason, allowing the library to safely use it on the Rust side of the code.

**V.4 Callback Manager**

A problem that arose from development came from the use of callback functions. When Rust wants to call a method in C# based on a pointer C# gives it, the library should be able to safely access that method. One issue comes from Rust operating in unmanaged memory and C# operating in managed memory. Managed memory tends to move around to save space due to a garbage collector. The garbage collector destroys objects no longer in use in managed memory and shifts other objects in memory around to fit better into memory. The issue comes from timing. In a theoretical scenario, if C# has a delegate, or temporary method, it can give that delegate’s pointer to Rust and say that it is a C method, allowing Rust to execute it. Between the time that Rust tries to execute that method and C# gives it that method, the garbage collector can move that delegate in memory. Then, when Rust tries to call that method, because it is no longer in that location in memory, a memory leak will occur and the program will halt. Our library still wants to be able to use callbacks as they’re very important to our asynchronous C# and Rust method architecture, and so we developed the Callback Manager.

It works by utilizing the fact that in managed memory, if an object is moved in memory, other objects in managed memory with references to it will still be able to see it. It also utilizes the fact that static objects and methods in managed memory will not move since they are never collected or considered by the managed memory’s garbage collector. Whenever the library has a delegate in C# it wants Rust to be able to call, it first stores that delegate into a dictionary and generates a key representing its location in that dictionary. Then, it gives Rust a method pointer to a static method that takes a key as an input and whatever outputs from Rust the delegate might need. Rust then executes and calls the static method in C# passing back the key it was given earlier and its outputs to the static method. The static method then uses the key to find the delegate in managed memory and executes that method using the other arguments it was given. This allows for the safe execution of delegates from unmanaged memory into managed memory without the risk of memory leaks.

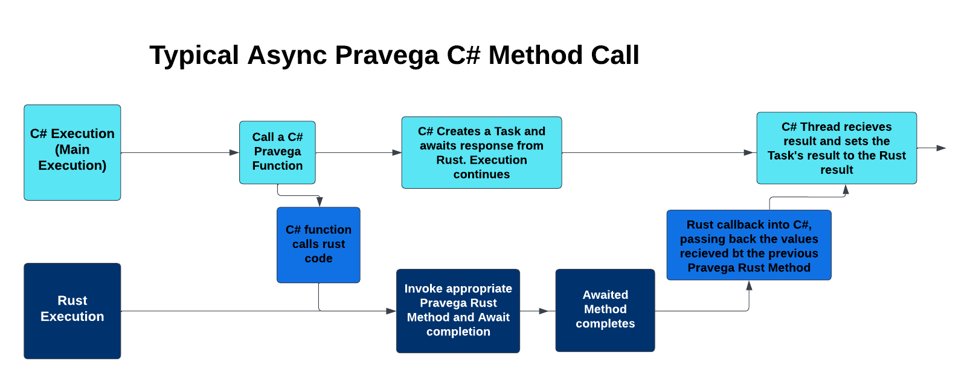
**V.5 PathGen**

An early problem encountered during development was the issue of C# not being able to find Rust code. The issue arose because of how C#’s execution was looking for the Rust code files and it was missing the files the library had it in. It would be cumbersome for a user to have to set environment variables on their computer so that the library could find the Rust code and be useable, so PathGen was developed. It works by setting the execution’s environment directory to be above our library so that the execution can find the Rust code. While not intended for the end user, it is meant more to help with development and helps testing.

**V.6 Asynchronous Communication Between Rust and C#**

C# and Rust have different forms of asynchronous architectures, but ideally the library should communicate between them through the parts they have in common. This came with the utilization of Rust futures and C# tasks. Rust futures can be thought of as asynchronous operations that Rusts promises will eventually and produce an output. Their completion can be awaited. C# tasks can be thought of as asynchronous operations that C# promises will eventually produce an output and that output can be manually set. Their completion can be awaited. With these two pieces being closely related, we developed this architecture for our project.

A Rust future can be used in Rust to manage an asynchronous method. When that method stops it will produce an output and that output needs to be picked up by C# at the time it completes. C# tasks can be manually set and so the architecture revolves around Rust operating asynchronously and once its output is produced, the output is then set as the output of the C# task.



In a typical asynchronous call, C# will initiate a call to Rust, following the same architecture as an FFI callback in most part. A Task is created in C# which will hold the output of the Rust method eventually. A delegate, or temporary method, is created that takes the outputs of the Rust method and sets it to the output of the Task. The delegate created is stored in the Callback Manager and the manager produces a key to access it later. To Rust, the arguments to the desired asynchronous method are inputted along with a static method pointer to the callback manager and the appropriate key. The method passed into Rust will invoke the earlier created delegate based on the key. Inside of Rust, a thread is created based on the Client Factory’s asynchronous runtime and the appropriate asynchronous Rust method is called on that runtime. While the asynchronous method is executing, the Rust function ends and C# resumes normal execution. Once the asynchronous function has completed, the output and key will be sent back to C# via the earlier given static method. The static method uses the key to find the delegate function it corresponds to and execute it, passing in the other arguments it received to it. The Task’s result is set to the output and the task is marked as complete. This solution utilizes the callback manager and the relationship between Rust futures and C# tasks to safely allow the communication between Rust and C# asynchronous operations. This relationship is one way, with Rust async being allowed to communicate with C# async, but not the other way around, though this architecture can likely be adjusted to do so as well in another project.

**V. Test Case Specification and Results**

**V.1. Testing Overview**

Our main objectives in testing were to verify that the functionality of the native Rust API was captured in our C# wrapper, and that the time difference between the two was not too large. An increase in time was to be expected, as that is just a cost of wrapping existing code. Testing was primarily done in NUnit tests. These allow for easy integration with C# code which is the basis for our project. The major testing categories as related to our project are:

* Unit Testing: NUnit tests start out with the most basic aspects of our project and gradually work their way up to the more complex parts.
* Integration Testing: This was tested along with our unit tests. Each unit test also tested the integration of the Rust code and the C# code as well as how the C# code is able to connect to the Pravega server.
* System Testing: Again, in order for NUnit tests to pass, the entire system also had to be running as desired. This means that like integration testing, System testing was done simultaneously with Unit testing.
* Functional Testing: All of the NUnit tests also tested to see if the desired outcome would be produced by each test. This ensured functionality was tested
* Performance Testing: We tested the time that it took for out C# Wrapper to perform the same process as the Rust API, and then compared the two times
* User Acceptance Testing: This was accomplished as a result of receiving feedback from our mentors in the form of weekly meetings and surveys about the performance of the project.

**V.2. Environment Requirements**

The testing can be run by downloading or cloning out GitHub repository, and then opening the solution file for our code base in Visual Studio. From there the tests can be initiated by right clicking on PravegaCSharpTestProject in the solution explorer and clicking run tests. For the tests to run the Rust code must be compiled into the PravegaCsharp.dll file, and an active local Pravega server needs to be running.

**V.3. Results**

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Time | Trait | Error |
| ByteReaderAvailableTest Passed Stale | 182 ms |  |  |
| ByteReaderCurrentHeadTest Passed Stale | 156 ms |  |  |
| ByteReaderCurrentOffsetTest Passed Stale | 147 ms |  |  |
| ByteReaderCurrentTailTest Passed Stale | 204 ms |  |  |
| ByteReaderReadTest Passed Stale | 191 ms |  |  |
| ByteWriterCurrentOffsetTest Passed Stale | 138 ms |  |  |
| ByteWriterFlushTest Passed Stale | 187 ms |  |  |
| ByteWriterResetTest Passed Stale | 220 ms |  |  |
| ByteWriterSealTest Passed Stale | 216 ms |  |  |
| ByteWriterSeekToTailTest Passed Stale | 225 ms |  |  |
| ByteWriterTruncateBeforeTest Passed Stale | 146 ms |  |  |
| ByteWriterWriteTest Passed Stale | 86 ms |  |  |
| ClientFactoryAsyncTimeTest Failed Stale | 83 ms |  | Expected: True But was: False |
| ClientFactoryClientConfigConstructorTest Passed Stale | 26 ms |  |  |
| ClientFactoryClientConfigConstructorTimeTest Passed Stale | 53 ms |  |  |
| ClientFactoryConfigRuntimeConstructorTest Passed Stale | 30 ms |  |  |
| ClientFactoryConfigRuntimeConstructorTimeTest Passed Stale | 126 ms |  |  |
| ClientFactoryConfigTest Passed Stale | 56 ms |  |  |
| ClientFactoryConfigTimeTest Failed Stale | 93 ms |  | Expected: True But was: False |
| ClientFactoryDefaultConstructorTest Passed Stale | 70 ms |  |  |
| ClientFactoryDefaultConstructorTimeTest Passed Stale | 57 ms |  |  |
| ClientFactoryHandleTest Passed Stale | 29 ms |  |  |
| ClientFactoryHandleTimeTest Failed Stale | 90 ms |  | Expected: True But was: False |
| ClientFactoryRuntimeTest Passed Stale | 27 ms |  |  |
| ClientFactoryRuntimeTimeTest Failed | 107 ms |  | Expected: True But was: False |
| ClientFactoryToAsyncTest Passed Stale | 26 ms |  |  |
| ConfigConstructor Passed Stale | 36 ms |  |  |
| ConfigControllerUriTest Passed Stale | 48 ms |  |  |
| ConfigMockTest Passed Stale | 25 ms |  |  |
| ConfigRetryPolicyTest Passed Stale | 50 ms |  |  |
| ConfigTlsEnabledTest Passed Stale | 25 ms |  |  |
| CustomStringCloneTest Passed Stale | 25 ms |  |  |
| CustomStringDefaultConstructorTest Passed Stale | 24 ms |  |  |
| CustomStringFromBigRustStringTest Passed Stale | 611 ms |  |  |
| CustomStringFromBigStringTest Passed Stale | 396 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ByteReaderConstructorTest Passed Stale (2) | 738 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ByteReaderSeekTest Passed Stale (15) | 517 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ByteWriterConstructorTest Passed Stale (2) | 607 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ConfigMaxConnectionsInPoolTest Passed Stale (3) | 30 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ConfigMaxControllerConnectionsTest Passed Stale (3) | 31 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ConfigTransactionTimeoutTest Passed Stale (3) | 32 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ControllerClientCreateScope Passed Stale (2) | 560 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ControllerClientCreateStream Passed Stale (2) | 575 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringCapacityTest Passed Stale (2) | 25 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringFromCustomStringTest Passed Stale (2) | 24 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringNativeStringAndConstructorTest Passed Stale (2) | 24 ms |  |  |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringRustStringAndConstructorTest Passed Stale (2) | 27 ms |  |  |
| RetryWithBackoffDefaultConstructor Passed Stale | 31 ms |  |  |

**Projects and Tools Used**

The main tools used for development were Pravega, Interoptopus, and NUnit. This project was designed to have no additional libraries as this is suppose to be compatible with all systems that can run .NET, so any extra libraries would work against that goal

|  |  |
| --- | --- |
| Tool | Purpose |
| Pravega | Provides code in Rust to call from C# and base our C# wrapper from. |
| Interoptopus (depreciated) | Helps convert code between C# and Rust such that C# can invoke methods from Rust as well as pass simple objects into Rust. No longer used explicitly for converting code, but instead for verifying the code we are implementing is converted correctly. |
| NUnit | This library is not essential for running our code, but helps us perform unit, integration, and system tests. |

For programming languages, we have only had to use C# and Rust since they are the basis of our project.

Diagram

Description automatically generated**VII. Description of Final Prototype**

Our final prototype has 2 main features. They are: 1) Wrappers in C# for ClientFactory, Byte, the event\_writer portion of Event, and miscellaneous shared features (ClientConfig, methods for transferring C# strings into Rust, etc.) and 2) a Rust .DLL library that the C# code can call functions from that mediate the C# and native Rust API. Below is a diagram for how this operates with the ByteWriter class.

The current functionality of each major module’s C# wrapper is as follows:

**Client Factory Wrapper Functions**

* **Initialize (ClientConfig? factoryConfig = null, TokioRuntime? factoryRuntime = null);**
  + Creates a new default instance for ClientFactory using the default Pravega configuration settings and a generated runtime.
* **TokioRuntime Runtime**
  + Getter for the ClientFactory object’s TokioRuntime
* **TokioHandle Handle**
  + Getter for the ClientFactory object’s TokioHandle
* **ClientConfig Config**
  + Getter for the ClientFactory object’s configuration settings
* **ClientFactoryAsync ToAsync()**
  + Returns the ClientFactory object’s async counterpart
* **Bool Initialized()**
  + Verifies that the client factory was previously initialized
* **Destroy()**
  + Sets the pointer to NULL
* **Task<ByteWriter> CreateByteWriter(ScopedStream writerScopedStream)**
  + Creates a new ByteWriter asynchronously
* **Task<ByteReader> CreateByteReader(ScopedStream readerScopedStream)**
  + Creates a new ByteReader asynchronously
* **Task<ReaderGroup> CreateReaderGroup(ScopedStream readerGroupScopedStream)**
  + Creates a new ReaderGroup asynchronously
* **Task<ReaderGroup> CreateEventWriter(ScopedStream readerGroupScopedStream)**
  + Creates a new EventWriter asynchronously

**Byte Writer Wrapper Functions:**

* **ByteWriter()**
  + Creates a new ByteWriter asynchronously
* **InitializeByteWriter (ScopedStream writerScopedStream)**
  + Initializes the ByteWriters IntPtr
* **Task<IntPtr> GenerateByteWriterHelper(ScopedStream writerScopedStream)**
  + Helps Initialize the ByteWriter IntPtr
* **Ulong CurrentOffset**
  + Getter for Rust byte\_writer current offset
* **Task<ulong> Write( List<byte> buffer)**
  + Writes bytes to a stream from a list of bytes
* **Task<ulong> Flush()**
  + Flushes the stream
* **Task<ulong> Seal()**
  + Seals the stream
* **Task<ulong> TruncateDataBefore(long offset)**
  + Truncates data at specific offset
* **Task<ulong> SeekToTail()**
  + Goes to the end of the segment
* **Task<ulong> Reset()**
  + Resets the internal reactor

**Byte Reader Wrapper Functions:**

* **ByteReader()**
  + Creates a new ByteReader asynchronously
* **InitializeByteReader (ScopedStream writerScopedStream)**
  + Initializes the ByteReader IntPtr
* **Task<IntPtr> GenerateByteReaderHelper(ScopedStream writerScopedStream)**
  + Helps Initialize the ByteReader IntPtr
* **Ulong CurrentOffset**
  + Getter for Rust byte\_reader current offset
* **Task<ulong> Seek(ulong mode=0, long numberOfBytes=0)**
  + Seek to a current offset
* **Task<byte[]> Read(ulong numberOfBytesRequested)**
  + Reads from a stream
* **Task<ulong> CurrentHead()**
  + Gets the current head of readable data
* **Task<ulong> CurrentTail()**
  + Gets the current tail of readable data

**Event Writer Wrapper Functions:**

* **EventWriter()**
  + Creates a new ByteWriter asynchronously
* **InitializeEventWriter (ScopedStream writerScopedStream)**
  + Initializes the EventWriters IntPtr
* **Task<IntPtr> GenerateByteWriterHelper(ScopedStream writerScopedStream)**
  + Helps Initialize the EventWriter IntPtr
* **Task<ulong> WriteRoutingKey(List<byte> buffer,String routingKey)**
  + Writes an Event with the specified routing key from a list of bytes
* **Void DropEventWriter()**
  + Drops the current EventWriter from the server
* **Task<ulong> EventWriterFlush()**
  + Flushes Event Writer Data
* **Task<ulong> Write(List<byte> buffer)**
  + Writes an event using list of bytes

**Additional Aspects:**

Several other features were created to help facilitate interoptability between C# and Rust. These features are:

* **ClientConfig:** A wrapper for the client\_config used in rust, used to set parameters of a ClientFactory
* **ControllerClient:** wrapper for the controller\_client Rust object. Allows a C# user to interact with a running Pravega stream
* **CallBackDelegateManager:** A program that uses dictionaries to store C# lambda functions as values with an ulong key. This allows a Rust function call to use a function pointer to invoke a lambda C# function that marks a Task as complete.
* **PathGen:** Program that is useful in development for finding the path of .Dll files.
* **Miscellaneous Shared Data Types:** several data types were created in C# to facilitate the transfer of data into a Rust palatable function call. One example is converting strings into UTF8 for Rust to understand.

**VIII. Project Delivery Status**

As of May 2, 2023, the project’s status is as follows: we have developed successful wrappers for the ClientFactory, Byte, and EventWriter modules. While this was not all the modules that we originally set out to complete, we were able to produce a basis for a fairly unresearched problem: wrapping async Rust Code in C#. We believe that we are one of the first to do so, and our project showcases the discoveries me made such as using a block\_on call to use a Rust runtime effectively, and making sure that all Rust wrapper functions have access to the same memory space as the allocated ClientFactory.

1. <https://github.com/WSUCptSCapstone-Fall2022Spring2023/dell-pravegaapi>

The above link is the GitHub web page on which the project is currently being stored. It can be used at its current state by simply cloning the repository and opening the VS solution file in the Project\_Code\_Base directory.

**IX. Conclusions and Future Work**

**IX.1 Limitations and Recommendations**

The biggest limitation of the project currently is that it currently is only able to wrap the Byte, ClientFactory, and EventWriter Modules. It’s missing the second half of Event, Sync, and Error Modules​. It’s also not as time efficient as the base Rust code. There is also no easy way to manage errors between the languages efficiently. We had to improvise by using Visual Studio to work with the C# side and Visual Studio Code for the Rust side, constantly flipping back and forth. Lastly, there is no security feature to make sure that Rust object pointers can’t be simultaneously accessed/deleted.

**IX.2 Future Work**

Going hand in hand with the limitations of our project, the future work left to be done is implementing/finishing everything mentioned in the above section. The remaining modules need to be implemented. This should be faster than starting from scratch, as we have found a successful method to wrap other aspects of the Rust code. The time measurements also need to be looked at, and improved, if possible. Locks need to be implemented on the C# side, so that if asynchronous tasks are being called on a singular Rust object they can be managed, and especially so that an object pointer can’t be deallocated while still in use. Finally, a more efficient way of dealing with errors should be developed, such as having both side log errors to a singular text file.

**X. Acknowledgements**

The foremost people that we would like to acknowledge are out mentors from Dell who have helped us in this project: Tom Kaitchuck, Sachin Joshi, Derek Moore, and Sesh Mandalika. They have us with countless tips from their experience in the industry and guided us in the proper way to work with Rust code and wrapping languages.

We would also like to thank the instructor leading the course, Ananth Jillepalli for guiding us through the final course in our college careers.

**Glossary**

* API: Application Programming Interface

**References**

[1] “Pravega concepts¶,” *Concepts - Exploring Pravega*. [Online]. Available: https://cncf.pravega.io/docs/v0.11.0/pravega-concepts/. (accessed Sep. 20, 2022).

[2] “What is safer\_ffi?,” *GitHub*, Sep. 19, 2022. <https://github.com/getditto/safer_ffi>

(accessed Sep. 20, 2022).

[3] “How we integrate Rust with C#,” *Structured Blog*, Sep. 18, 2018.

https://blog.datalust.co/rust-at-datalust-how-we-integrate-rust-with-csharp/ (accessed

Sep. 20, 2022).

[4] R. Biedert, “ralfbiedert/interoptopus,” *GitHub*, Sep. 18, 2022.

<https://github.com/ralfbiedert/interoptopus>

[5] “Introduction - Rust By Example,” *doc.rust-lang.org*

<https://doc.rust-lang.org/beta/rust-by->example/index.html (accessed Sep. 20, 2022).

[6] “Tour of Rust - Let’s go on an adventure!,” *tourofrust.com*. <https://tourofrust.com/> (accessed Sep. 20, 2022).

[7] BillWagner, “Introduction to C# - interactive tutorials,” *learn.microsoft.com*. https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tutorials/ (accessed Sep. 20, 2022)

[8] “Crate Pravega Rust Documentation,” https://docs.rs/pravega-client/0.3.1/pravega\_client/ (accessed Sep. 16, 2022).

[9] “Supported APIs · pravega/pravega-client-rust Wiki,” *GitHub*. https://github.com/pravega/pravega-client-rust/wiki/Supported-APIs (accessed Sep. 28, 2022).

[10] “Rust client for Pravega,” *GitHub*, Aug. 01, 2022. <https://github.com/pravega/pravega-client->rust/blob/master/book/src/Rust/ByteClient.md (accessed Sep. 28, 2022).

[11] “Transactions - Exploring Pravega,” *cncf.pravega.io*. https://cncf.pravega.io/docs/nightly/transactions/ (accessed Sep. 28, 2022).

[12] “Client APIs - Exploring Pravega,” *cncf.pravega.io*. https://cncf.pravega.io/docs/nightly/javadoc/ (accessed Sep. 28, 2022).

[13] “Fundamentals of Garbage Collection, “ learn.microsoft.com.

<https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals> (accessed Nov. 15, 2022)

[14] “Interface Client Factory,” *pravega.io.* <https://pravega.io/docs/v0.6.1/javadoc/clients/index.html?io/pravega/client/ClientFactoryhtml> (accessed Nov. 15, 2022)

[15] “Unit testing C# with NUnit and .NET Core,” *learn.microsoft.com.* <https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit> (accessed Nov. 15, 2022)

**Appendix**

1. **Team Information**

Members:

John Sbur

Brandon Cook

Samuel Lopez

Team Name:

The Pravegateers

1. **Example Testing Strategy Reporting**

Unit Test Example:

**Client Factory Default Constructor.**

Corresponding Requirement: ClientFactory Module. Functional Requirement

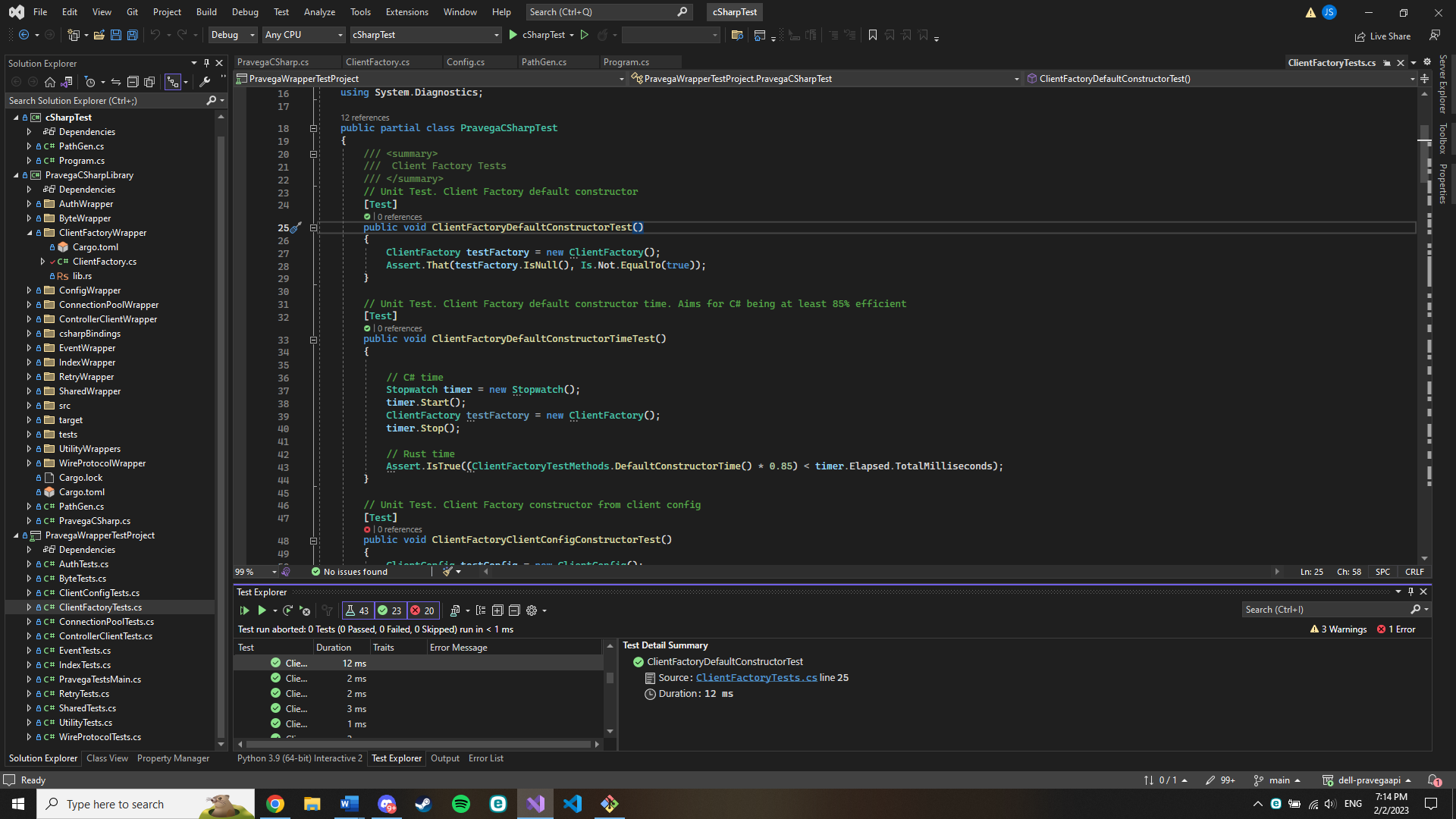
Results:

Expected Result: Creates a client config with constant predefined settings

Observed Result: Created a client config with said predefined settings and stored a reference to it in C#

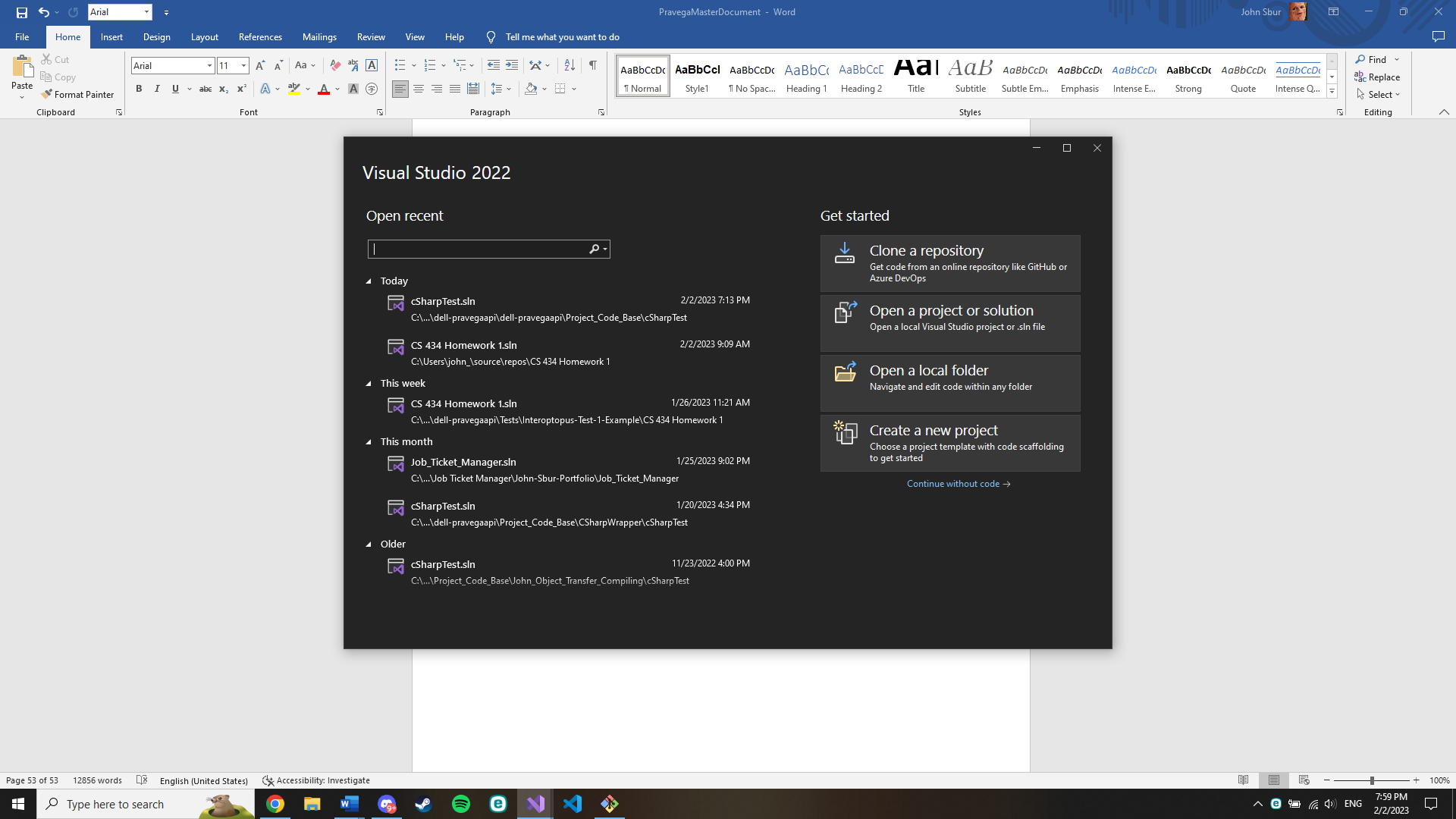
Test Result: Pass

Test case requirements: ClientFactory.cs, client\_factory\_wrapper.dll, machine that runs C# on .NET 6.0



**Replicating**

Replicating is set up to not be difficult. You can do so on any device with .NET 6.0 installed that can run Microsoft Visual Studio Community. The software can be found here: <https://visualstudio.microsoft.com/vs/community/>. Once installed with .NET 6.0, open and you will be greeted by a screen similar to this.



Select “Open a project or solution” and in the repository downloaded from our GitHub, there will be a .sln file that can be opened and have all our tests ran on. Link to our GitHub: <https://github.com/WSUCptSCapstone-Fall2022Spring2023/dell-pravegaapi/tree/main/Project_Code_Base/cSharpTest/PravegaWrapperTestProject>.

The file you are looking to open is “[cSharpTest.sln](https://github.com/WSUCptSCapstone-Fall2022Spring2023/dell-pravegaapi/blob/main/Project_Code_Base/cSharpTest/cSharpTest.sln).” The library used for testing is called NUnit. <https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit>.

1. **Project Management**

A typical weekly schedule will look something like this for our team:

**-Monday:**

Our team meets over Zoom, Discord, or in person to discuss details about the work that needs to be done during the week. Distribute workloads and make sure everyone understands what each person is doing and how it relates to them. Begin working on each person’s assigned workloads for the week.

**-Middle of the week:**

Continue working on our assigned workloads, making sure to let team members know through Discord when something is completed. If a member finishes early and has extra time in their schedule, they continue to work by helping other members who haven’t finished their work.

**-Friday Meeting:**

Every Friday at 4:15 p.m. our team meets with the Dell Pravega team to discuss what we accomplished during the week, what went well, what we have questions on, and what we are projected to do during the following week. We allow feedback to create tasks for the Agile Model backlog and conclude the meeting. Afterwards, our team talks about how the meeting went, discuss what the next week will look like, and sign off for the week.

This workflow has worked for our team over the previous semester and this semester with the only difference being that the Dell Pravega team meeting occurred on Thursday and the team meeting occurred on Friday. None the less, it’s vital to have both meetings to make sure that the Dell team is informed and our team is on the same page while working through our assigned workloads.

Progress has been steady and when we hit a roadblock in one area, we have been able to allocate our time efficiently to keep the project moving while tending to the damage.