Pravega Client Library for .NET

Sponsored by: Dell Technologies

Icon

Description automatically generated

**By: The Vegateers**

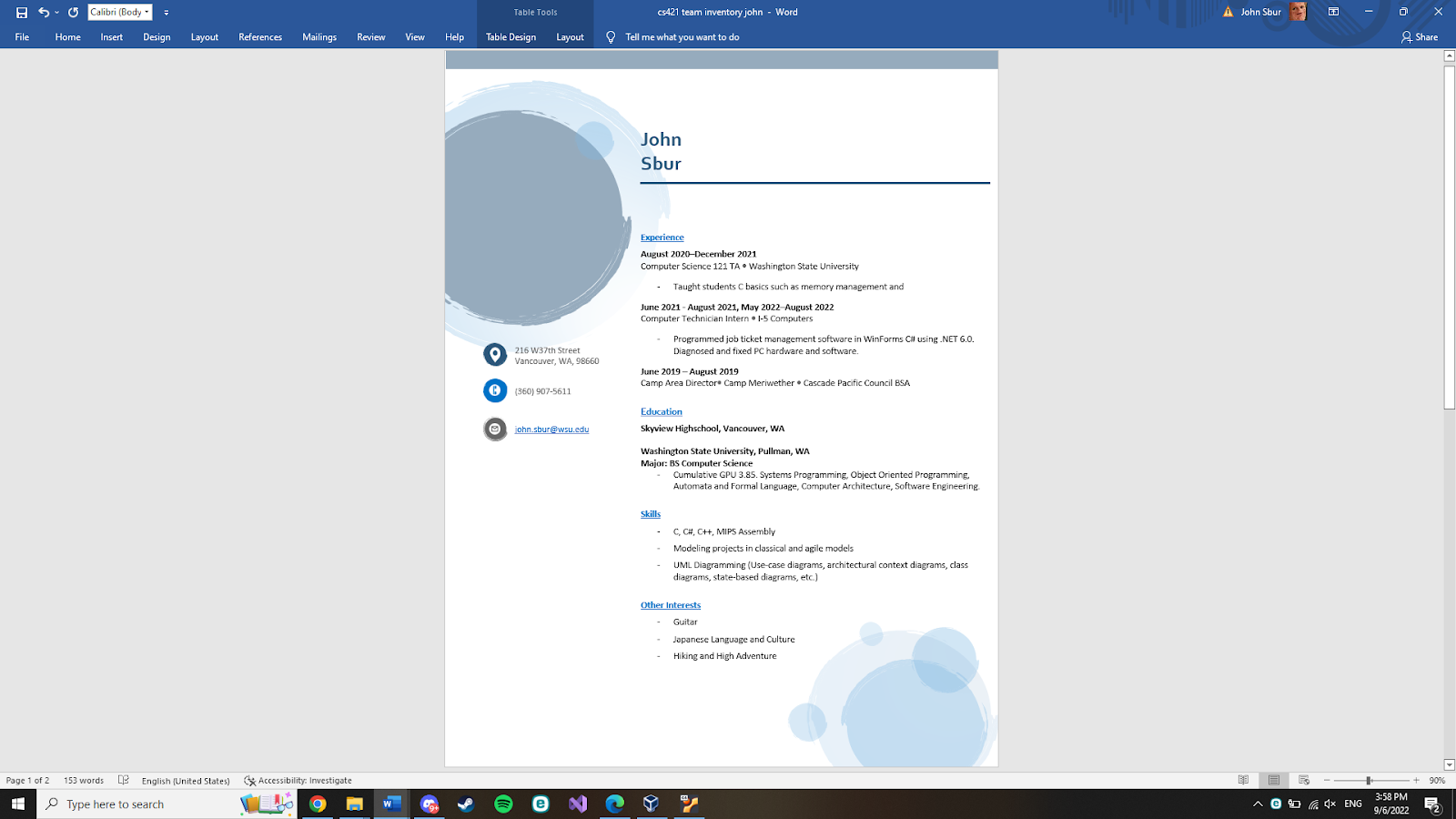
Sbur, John

Lopez, Samuel

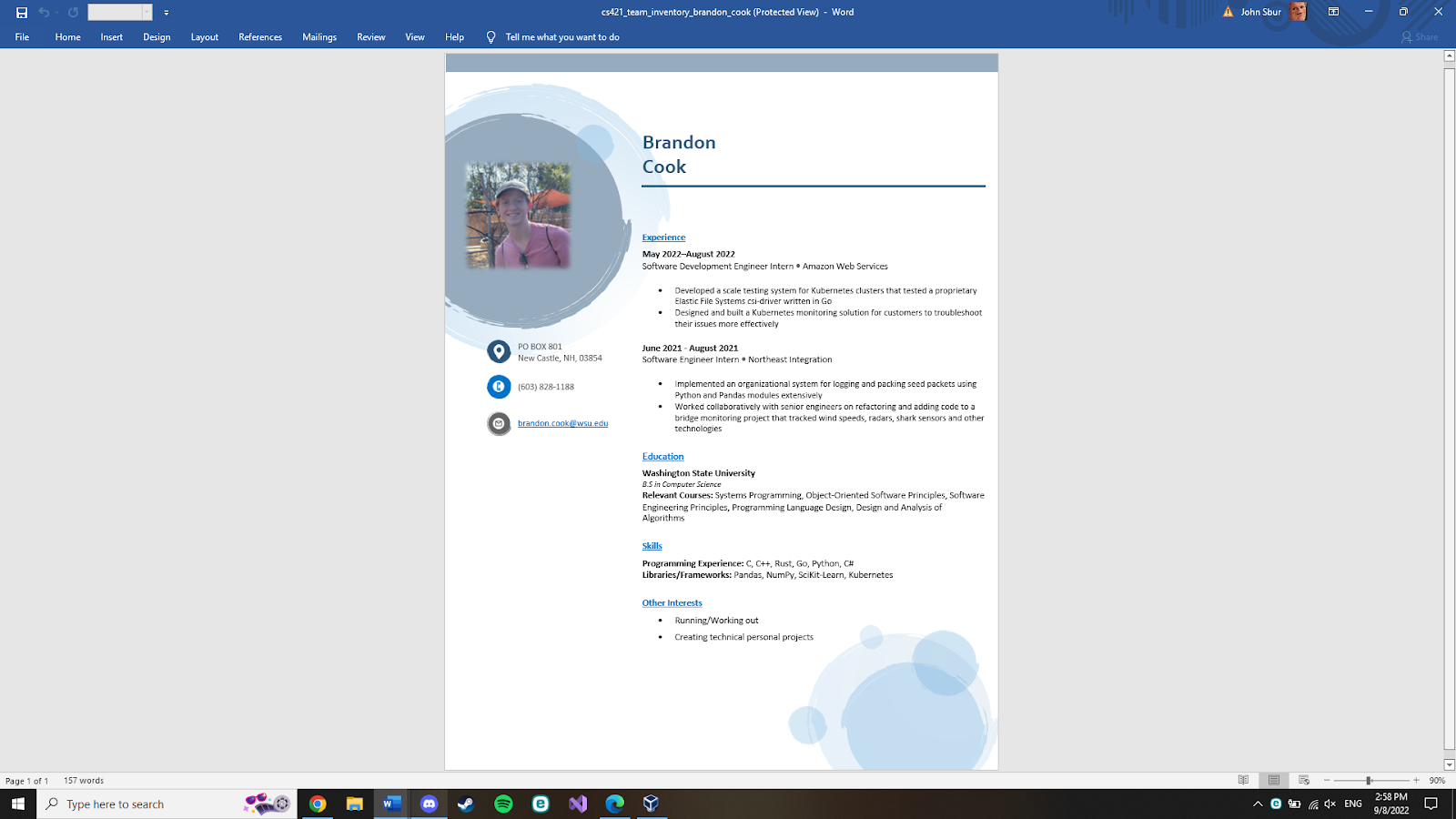
Cook, Brandon

**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 |
| Project Solution Approach | 16 |
| Introduction | 16 |
| System Overview | 16 |
| Architecture Design | 17 |
| Overview | 17 |
| Subsystem Decomposition | 18 |
| Byte | 18 |
| Event | 19 |
| Client Factory | 29 |
| Sync | 30 |
| Data Design | 33 |
| Byte | 33 |
| Event | 33 |
| Index | 36 |
| Client Factory | 36 |
| Sync | 37 |
| Error | 38 |
| Testing and Acceptance Plans | 39 |
| Introduction | 39 |
| Project Overview | 39 |
| Test Objectives and Schedule | 39 |
| Scope and Purpose of Document | 39 |
| Testing Strategy | 40 |
| Test Plans | 42 |
| Unit Testing | 42 |
| Integration Testing | 42 |
| System Testing | 42 |
| Functional Testing | 42 |
| Performance Testing | 43 |
| User Acceptance Testing | 43 |
| Environmental Requirements | 43 |
| Test Results | 44 |
| Module Tests | 44 |
| Client Factory |  |
| Byte |  |
| Event |  |
| Index |  |
| Synchronizer |  |
| Non-Module Tests |  |
| Authentication Wrapper |  |
| Config Wrapper |  |
| Connection Pool Wrapper |  |
| Controller Client Wrapper |  |
| Retry Wrapper |  |
| Shared Wrapper |  |
| Wire Protocol Wrapper |  |
| Utility |  |
| Table of Test Results | 53 |
| Project and Tools Used | 54 |
| Final Prototype | 54 |
| Description | . |
| Acknowledgements | . |
| Project Delivery Status | . |
| Future Work | . |
| Glossary | . |
| References | . |
| Appendix | . |
| Team Information | . |
| Example Testing Strategy Reporting | . |
| Project Management | . |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |





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 will be programming in 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 will be programming 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 shall be 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 will mean is that we will 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 shall be 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 plan on using something similar to CodeCov, an open-source program that analyzes branches and paths code can take that could fail or pass. Using this, we can help verify the integrity of the code and be sure that most branches are covered. Our goal is 85% coverage through all branches of code.

**[Open Source]**

This project shall be 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 shall be 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 shall 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 should 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 shall use 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 shall be 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 and so that it can be built upon easier when each step and function is explained in detail. Something similar to StyleCop, a program in C# designed to enforce C# naming conventions and documentation conventions, will be used to enforce this.

**[The System]**

This system shall be seamless for C# users. The use of this library should be as any other C# library. For people not familiar with Pravega, it should 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 shall also handle 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 shall log errors through Rust and send them through C#. This idea behind this is so that a programmer that encounters an error while using the library will still be able to find errors that were caused through the Rust library in C# instead of generic exceptions being thrown.

Finally, this system shall also uphold all of Pravega’s features. The idea is that this library created from this project would be 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 supposed to be a near-perfect mirror of what Pravega provides in Rust already. The goal is for this wrapper to operate time-wise at 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 planned, the underlying assumption is that the project vision will not change and the end goal will remain the same throughout development. However, should the initial plan altar through development, the Agile project management method we have implemented should prepare our 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. Methods are therefore to be kept simple and applicable to many situations and documentation will be written throughout the library to minimize the potential impact. A topic discussed already were some potential features we are prepared to implement if requested are security measures for encrypted data transfer as well as using LINQ within the C# framework. These have been mentioned as features to add if time allows it, especially LINQ because of its powerful capabilities.

One potential risk our team is keenly aware of is the choice of framework for building this C# library. The framework implemented will transfer the Pravega Rust code into C#, however each framework has its own bugs and specifics. If we choose the wrong framework, it could lead to disaster later down the road if something cannot be implemented, which would require an entire overhaul of the work up until that point, costing time. Therefore, our team is carefully considering each framework with the help of our clients who have experience in this field.

**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 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.

**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. Modules that are implemented are client factory and byte. Each of these modules had their own complex data structures taken into consideration while wrapping as these data structures.

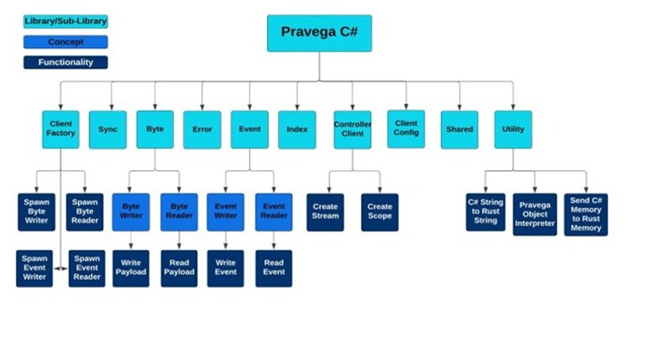
The way this project is designed to convert the modules and their content through the wrapper is by using an FFI, a mechanism that lets one programming language use code from another, to generate the code in C and then generate it into C#. Initially, we used Interoptopus to help with creating an FFI architecture, but later in development it became easier to generate our own. The advantage of a wrapper like this is that since it’s simply converting the code to C# with restrictions we put in place, 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. In addition, when converting the functions through an FFI doesn’t work or the output isn’t performing as well as our team wants, we may have the system use manually wrapped functions that we write. In either case, the system will go from Rust down to a C layer that is either generated or created manually. Since both Rust and C# recognize C, we can go from C to C# completing the transition.

Considerations made while designing this software is module dependance on each other as certain Pravega modules are coupled other modules to function. As mentioned earlier, Rust data structures need to convert over well to C# to maintain 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. One system is it coupled to is .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 and one of the main selling points of Pravega is its ability to handle multiple readers and writers at the same time while not losing time like other stream managers do. Without the ability to do synchronous or asynchronous tasks there is little reason to use this system in the first place over other stream managers that have this functionality implemented.

**Software Design**

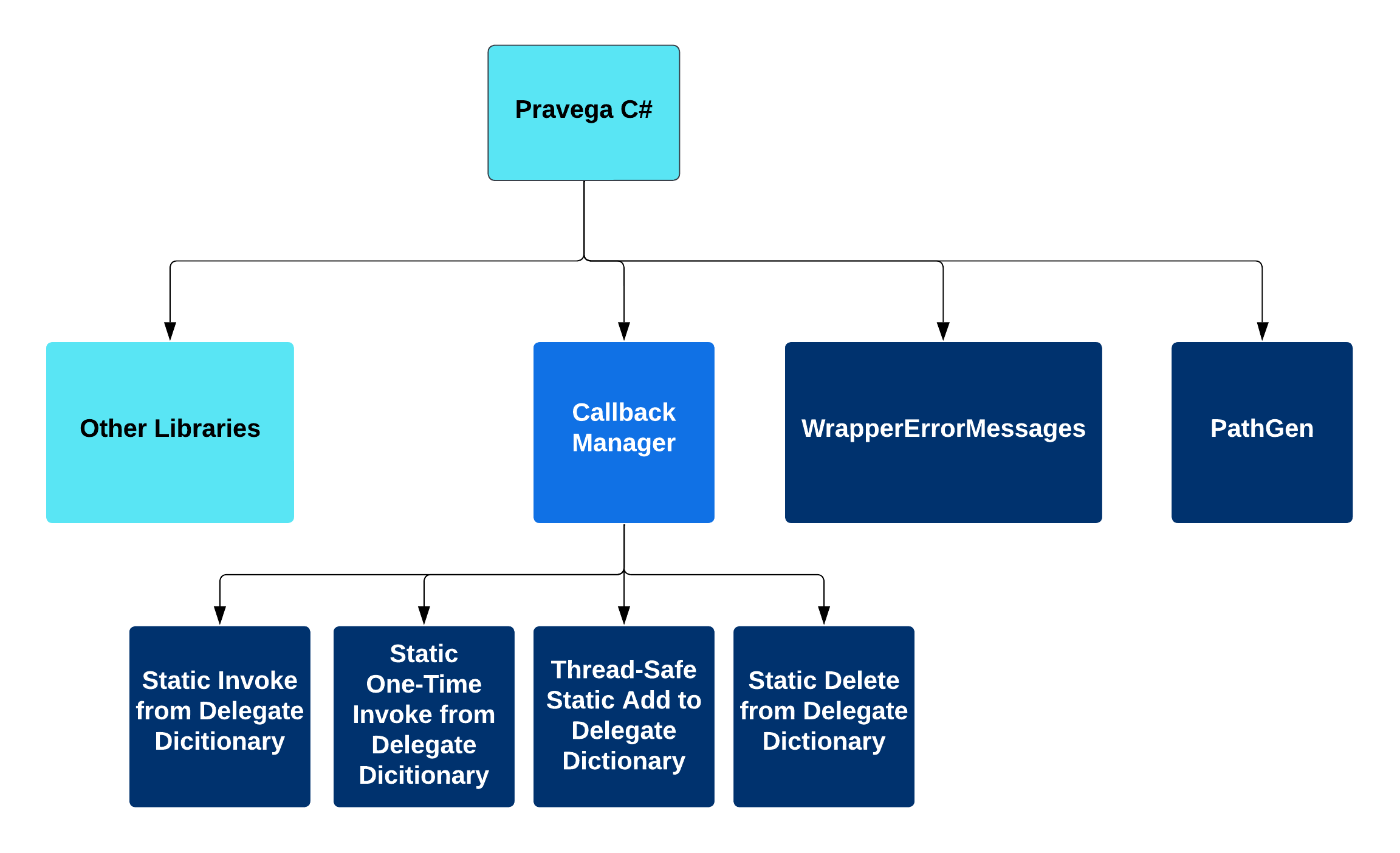
**I. Library Layout**



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

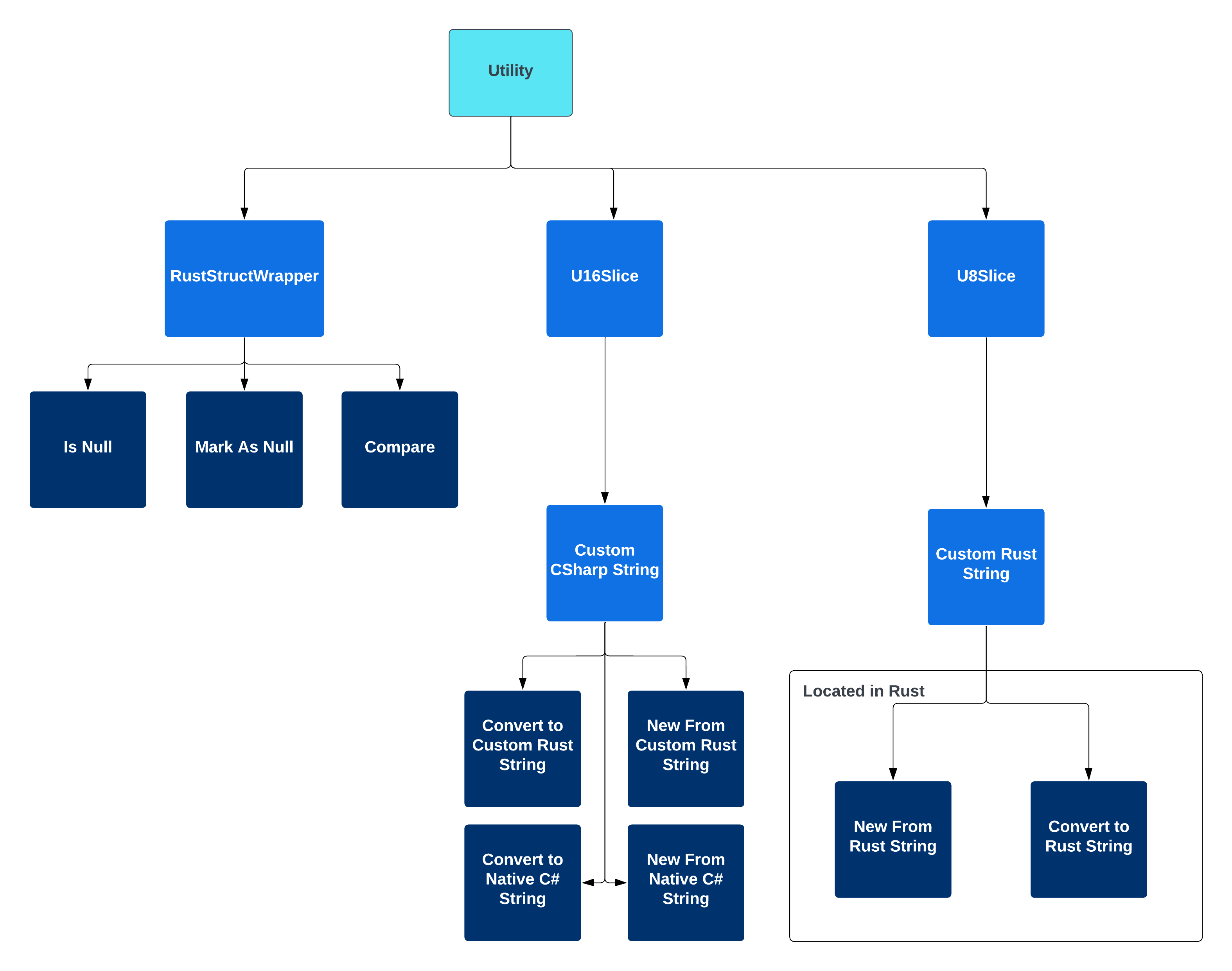
-Description: A technology developed for allowing safe communication between unmanaged languages and managed languages. This manager holds delegates, or temporary anonymous methods. Then, it allows Rust, operating in unmanaged memory, to call specific static methods in C#, which are in managed memory 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

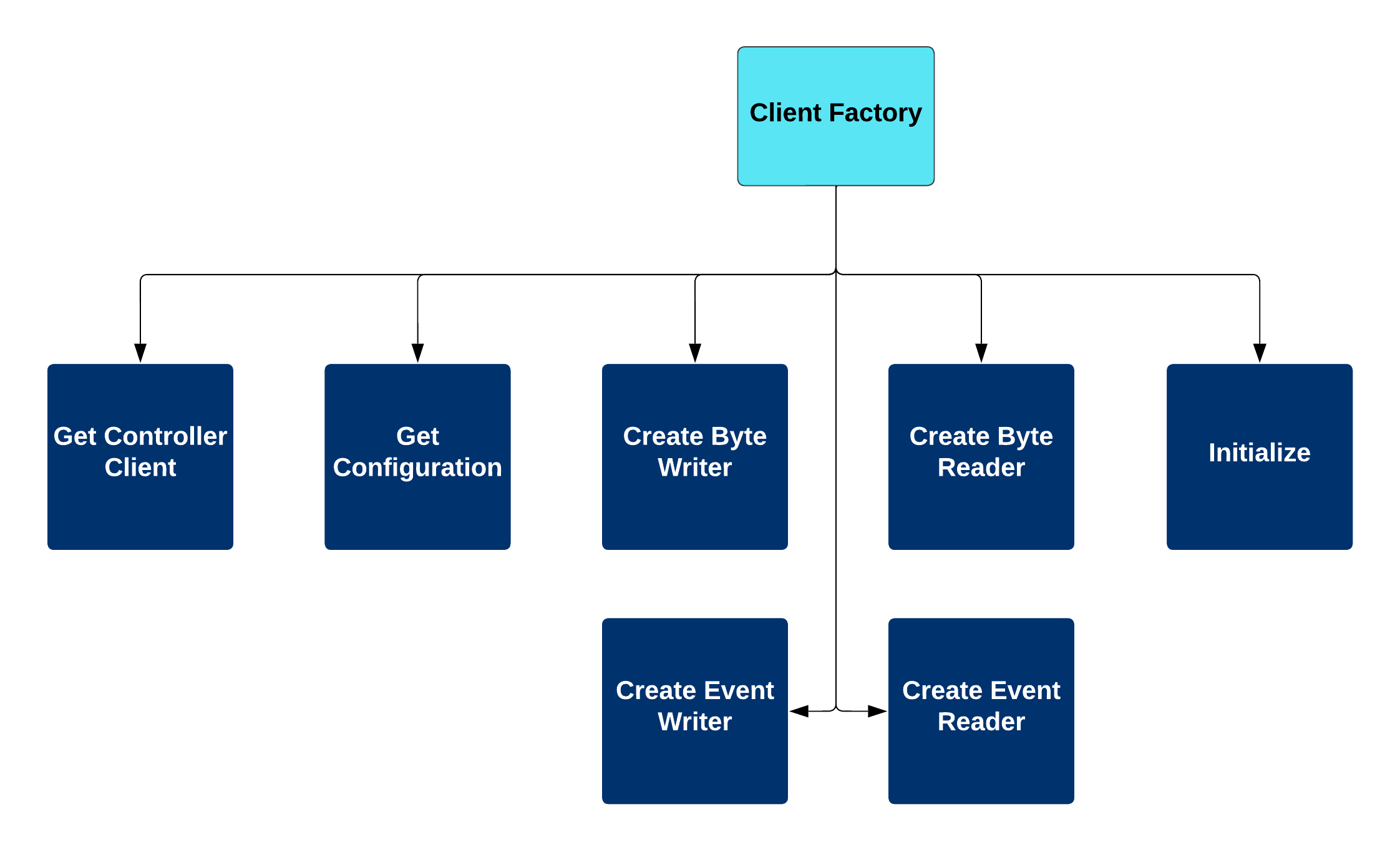
-Description: Allows for the comparison of two objects in Rust in C#. Tells the developer whether the objects are the same or not

**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.

**[ClientFactory]**



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

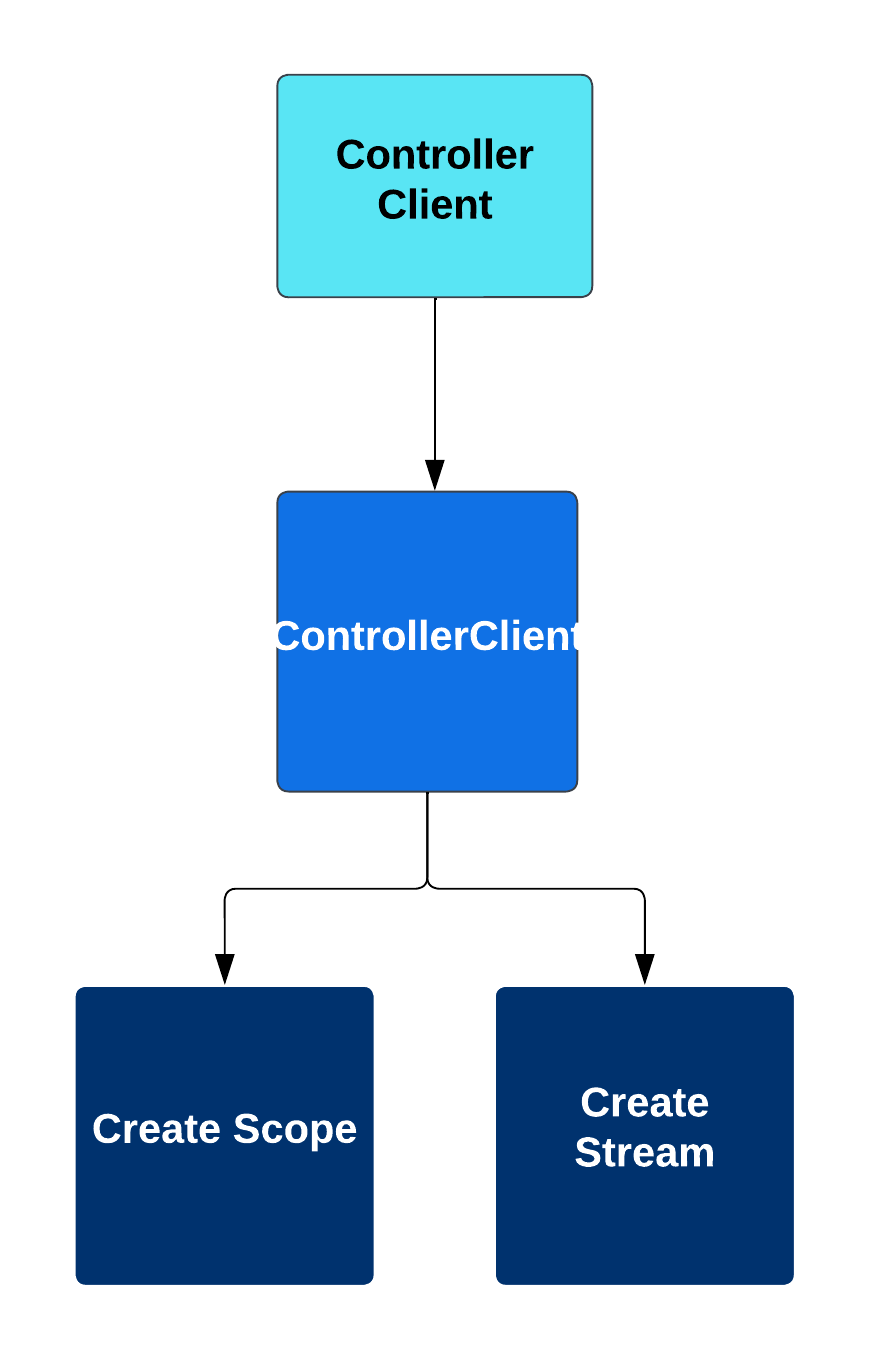
-Description: Allows a developer to get client factory’s controller which helps with the creation and modification of streams and scopes on the client factory’s Pravega Server.

**Service: Get Client Configuration**

-Provided to: C# Developer

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

**[ControllerClient]**



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

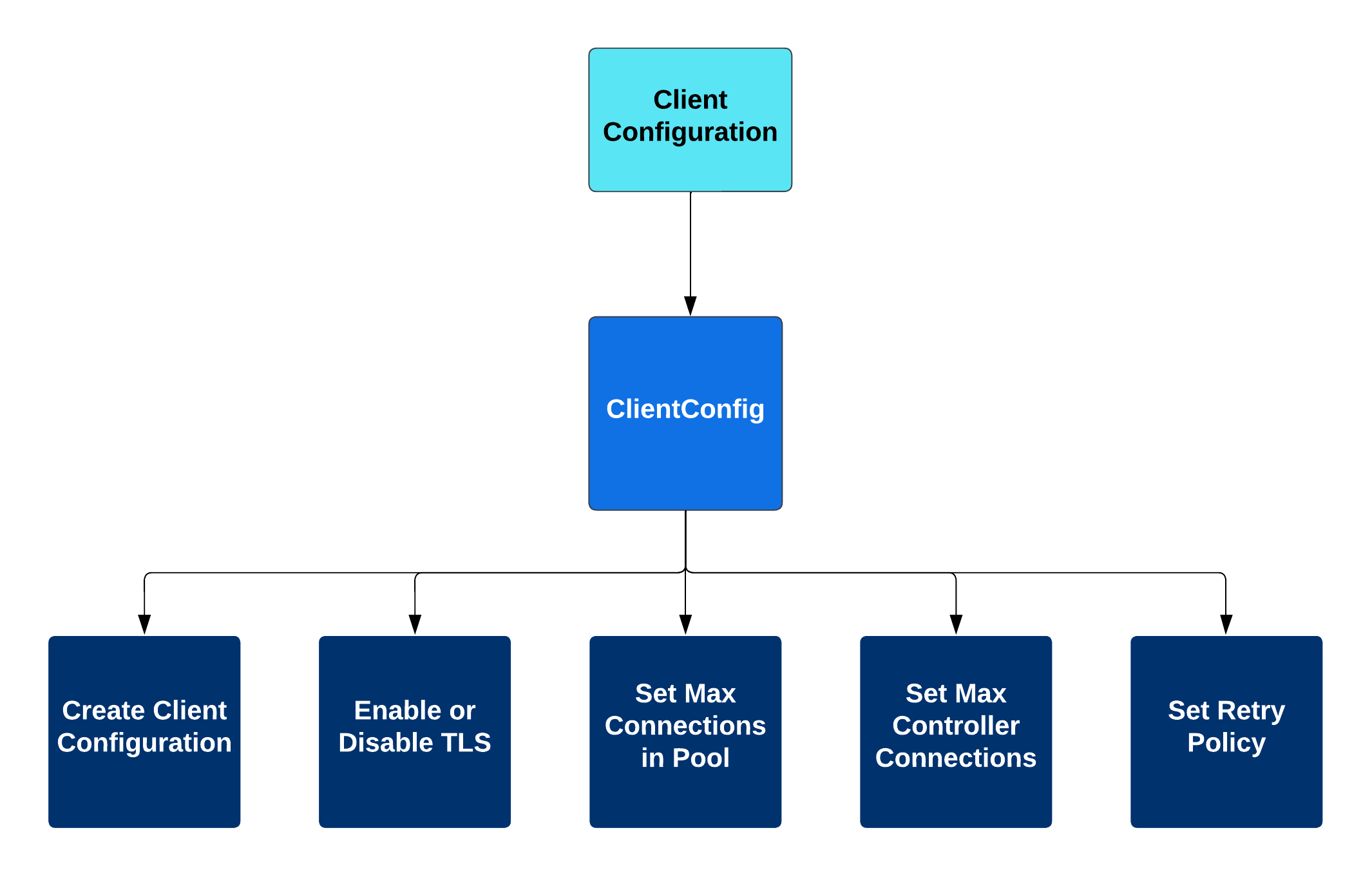
-Description: Allows the creation of scopes, environments to hold streams in, on a Pravega server.

**Service: Create Stream**

-Provided to: C# Developer

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

**[ClientConfig]**

****

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

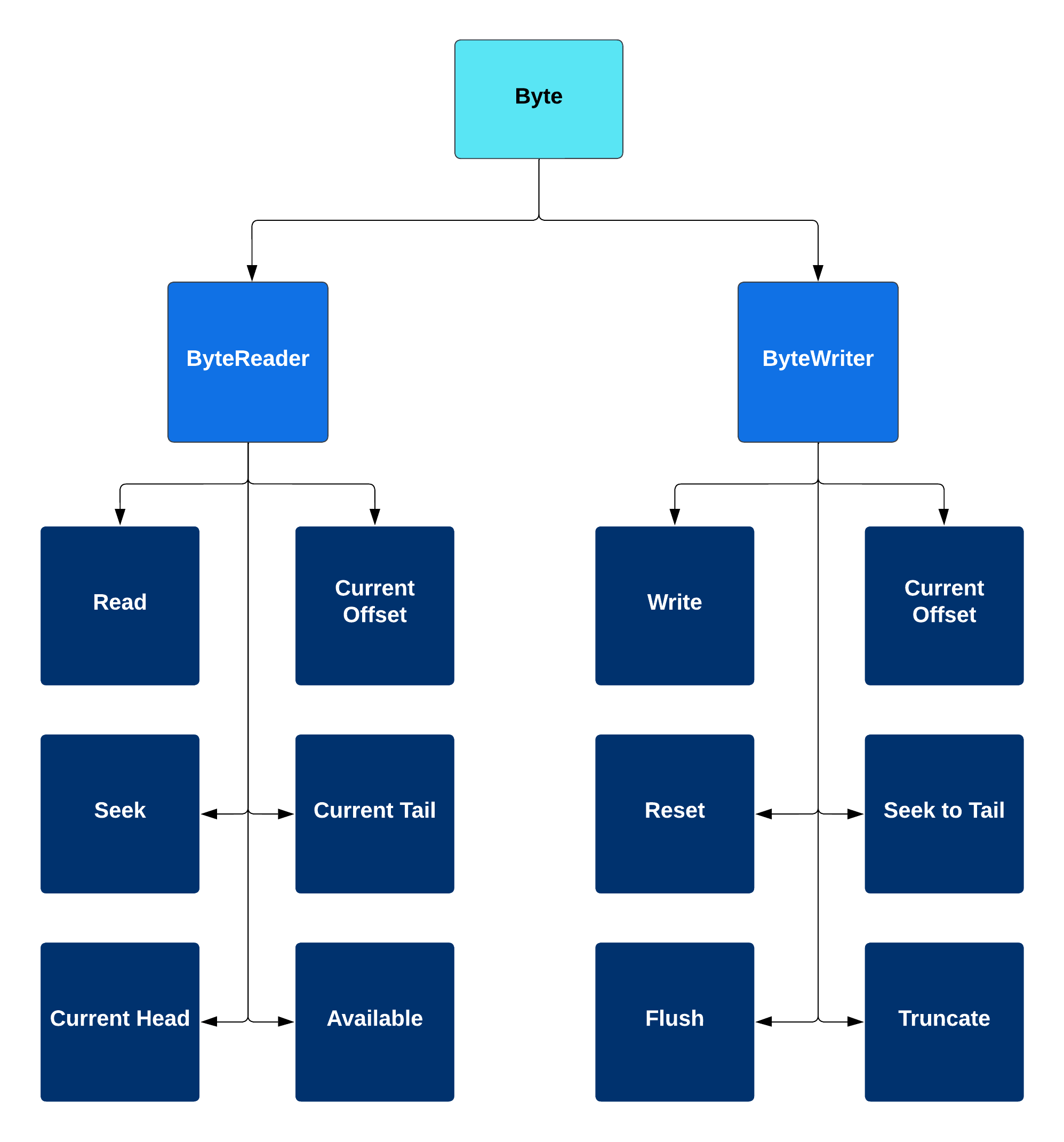
-Description: Allows the setting of how much time needs to pass for a server transaction to be considered a failure and timeout.

**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**

The EventReader module is how the user can read events from a Pravega stream. It is spawned via a ReaderGroup. It should be noted that the only way for a programmer to create an EventReader is through a ReaderGroup. It works by creating a SegmentSlice, which event data can then be read through.

It has the functions releaseSegment which releases a partially read segment slice back to the event reader, releaseSegmentAt which is similar but also indicates the offset, readerOffline which marks the reader as offline and passes its segments to other readers in the ReaderGroup, releaseSegmentFromReader which releases a provided SegmentSlice and marks it as unassigned, acquireSegment which returns a segmentSlice, fetchSuccesors which looks for a successor to a segment where an error occurred, assignSegmentsToReader which acquires newer segments for the reader, initiateSegmentReads which initiates a task to read from newly assigned segments, addDatatoSegmentSlice which appends data to the SliceMetaData, and getSuccessors which fetches the successors for a given segment.

**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.

It has the functions writeEvent which writes an event and generates a routing key, writeEventByRoutingKey which writes an event with an associated routing key, flush which flushes data, and clearInitialCompleteEvent which clears completed events from the flush queue.

**[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 Library]**

**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 Library]**

**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.

**[ControllerClient]**

**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.

**[ClientConfig]**

**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 Library]**

**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 Library]**

**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. It has functions init\_reader, release\_segment,release\_segment\_at,reader\_offline, release\_segment\_from\_reader,acquire\_segment, fetch\_succesors, assign\_segments\_to\_reader, initiate\_segment\_reads, add\_data\_to\_segment\_slice, and get\_successors.

Class EventWriter:

* This is the class used to write events.It has functions write\_event, write\_event\_by\_routing\_key,writer\_event\_internal, flush, and clear\_initial\_complete\_events. It holds a pointer to a rust eventWriter.

**[Shared Library]**

**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**

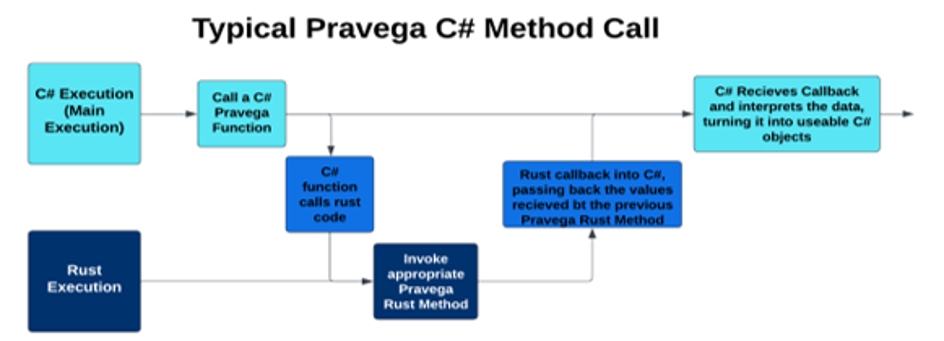
Our team went through many designs when asking the question of how to convert Rust objects into C# or how to represents Rust objects in C#. One of the first solutions was having all data in C# representing some form of the Rust data and then when we wanted to communicate between the two languages, we 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.

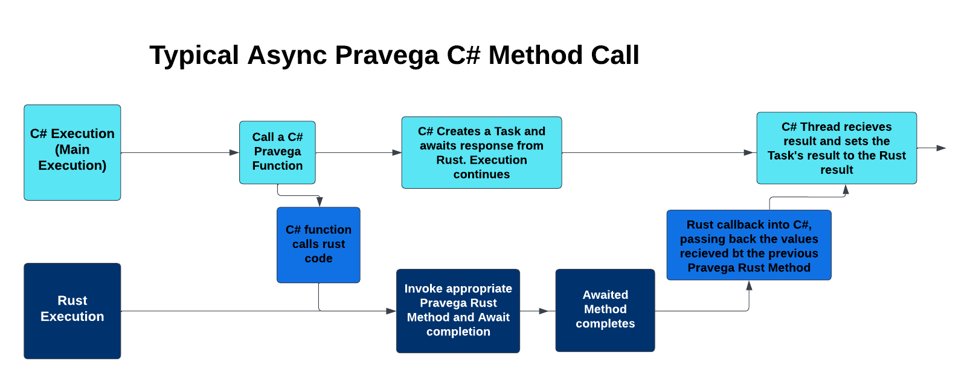
**V.5 PathGen**

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.

**Testing and Acceptance Plans**

**I. Introduction**

**I.1 Project Overview**

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 [N]. 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 [1].

**I.2 Test Objectives and Schedule**

Our general testing approach is to first test how different Rust features transfer over into C# via wrapping with Interoptopus. Then, using those results we will create wrappers for the major modules of the Pravega Client, and then test their functionality, performance, integration, and user acceptance. To do so, we will need access to our code, as well as Pravega, both locally and potentially to a server.

Our current schedule is set so that by November 9, 2022, we should be finished working with Rust components, and by November 16, 2022 we will be done testing the Rust components. This means that we will have code and documentation that shows how to wrap these components in C#. By November 27, 2022, we would like to have some of the smaller Pravega components finished, and finish testing them by December 4, 2022. At the end of this project we should have a working DLL that contains the wrapped code for C#.

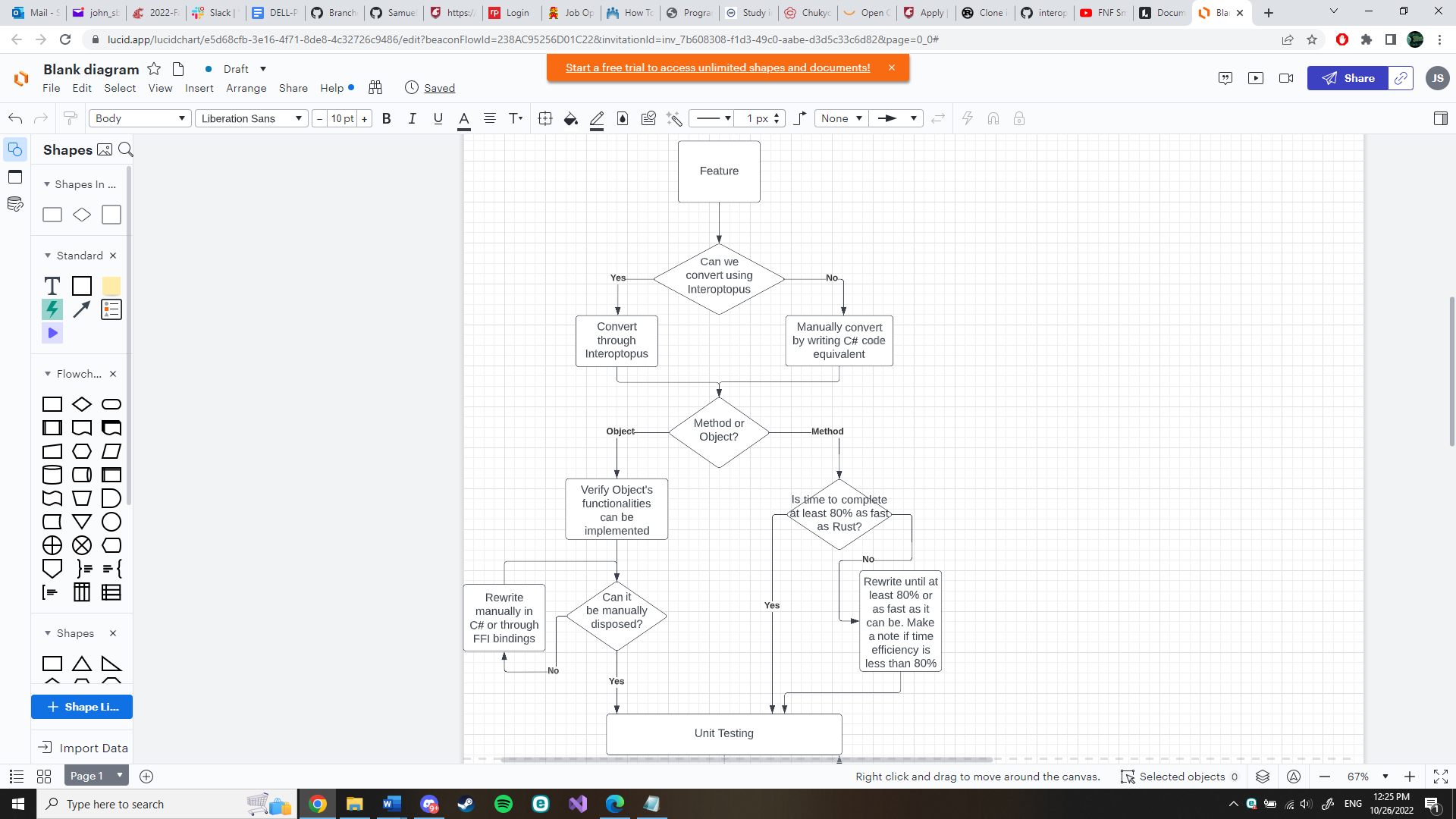
The major functionalities that we would like to test are being able to read and write events, being able to write, commit, and abort transactions, being able to read and write bytes, being able to synchronize states and tables, and being able to manage streams.

**I.3 Scope and Purpose of Document**

The purpose of this document is to outline the goals of the project, our solution approach, and how we will test our final version of the project. By reading the document, a user should be able to understand the end goals of the project, and the thought process behind out design decisions. The document contains an introduction to the project and its team members, our solution approach, and the requirements and specifications.

**II. Testing Strategy**

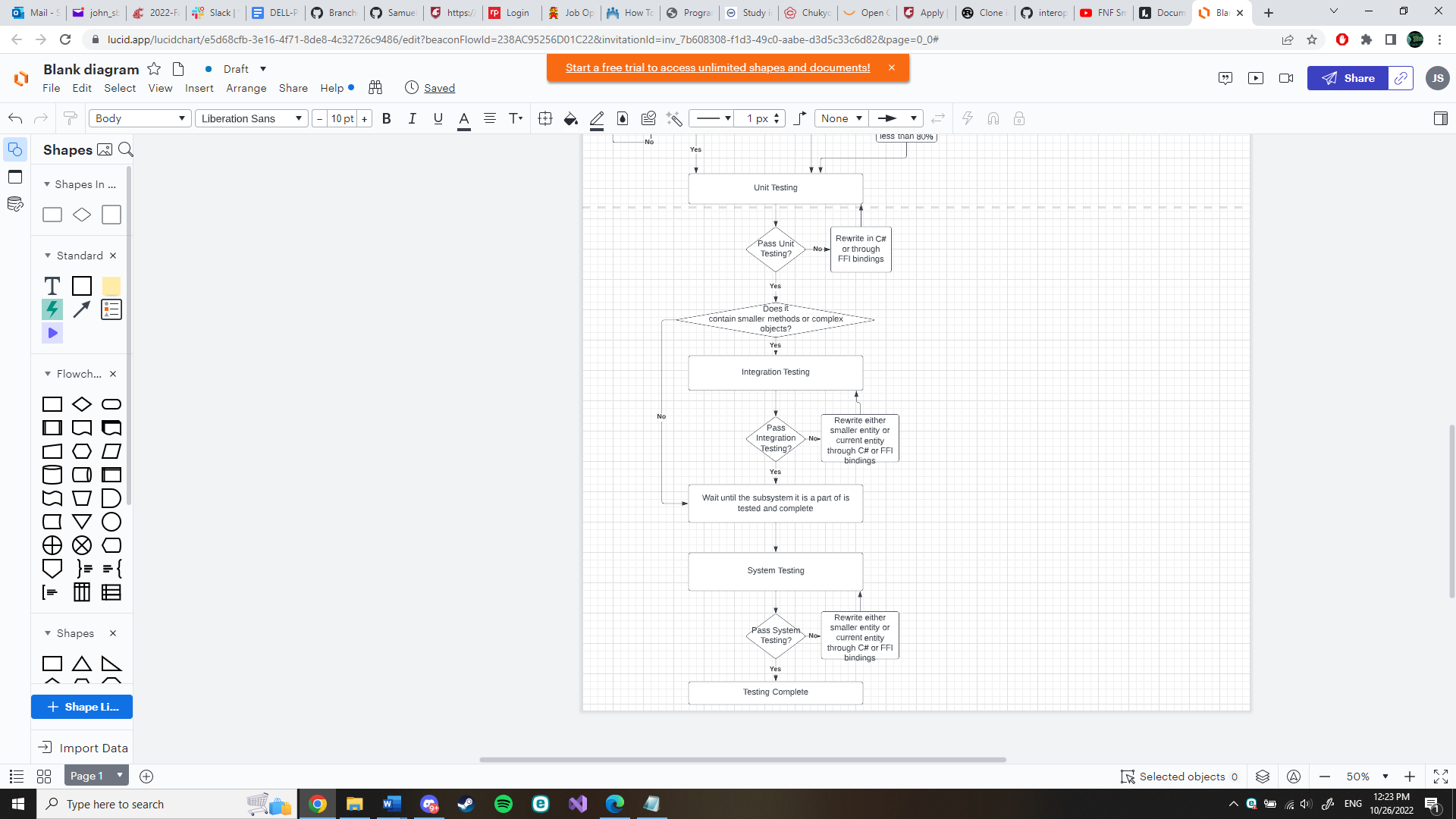
This project is about seeing how well features and functionalities in Pravega from Rust transfer from Rust to C#. As such, our testing involves seeing how well these features transfer over well. Features can be split into two categories, objects and methods.

The objects in Pravega’s Rust client such as structs and smaller components need to be able to be migrated over or have an equivalent form. In addition, because C# has a different memory management system than Rust, we need to make sure that time and memory are managed efficiently when migrated over to Rust. One of the big things is garbage collection in C# and the absence of that in Rust. When objects are no longer used in C# they are sent to a garbage collector to be disposed of later while in Rust it’s instantaneous[13]. To mimic the Rust’s functionality, we need to make sure that objects moved over are manually disposable in C# so that they can skip the garbage collection process.

The method in Pravega’s Rust client are the functions tied to Pravega’s objects and standalone methods. Besides unit testing, integration testing, and system testing these methods, we also want to make sure that moving from Rust to C# doesn’t impact the method’s time efficiency.

Below is a diagram showing how testing will take place for each feature.

Clarifications:

80%-time efficiency means that in a normal case, i.e., functions as expected with “normal” or “common” inputs, the time to complete compared to when those same tests are run on the Rust API will lose at most 20% of the time. It is expected that time will be lost in the layer of transitioning from Rust to C#, but it’s important to make sure that the time loss is not too extreme or the appeal of Pravega on C# will lessen.

Determining whether a functionality of an object can be implemented means that if a user were to use the object in Rust and that object is able to be cloned for example, then that object should also be able to be cloned in C#.

Documentation:

As testing progresses, test cases and exceptions from normal behavior will be documented alongside the tests. In the case that behavior seen is normal, then documentation will be located inside a code example of a test and when an exception happens, a note of it will be left in the feature’s testing folder.

Continuous Integration:

This project’s tests and code base will follow a Continuous Integration model, meaning many different branches will be created and merged over time as features are made and tested. This was chosen since the project is built on top of each other in blocks and many features need to be implemented and merged for group members to use before proceeding to more complex features. In addition, with the open-source nature of the project, this gives the potential of community feedback that can be integrated into the project’s backlog.

**III. Test Plans**

**III.1 Unit Testing**

Our general strategy for unit testing is to start with the basic components in a module, and then work our way up. The design of the client code in Rust is very layered, with different parts of the code building on top of other parts of the code. Our wrapper will attempt to mirror this approach. For example, ClientFactory can be used to spawn an EventReader, which will in turn use smaller structs to read data from a Pravega stream[14]. We will start with the basic structs in event reader, and then work our way upward in testing. We will likely design multiple unit tests per module, and move on to the next once all tests are passed.

**III.2 Integration Testing**

For integration testing, the plan is to test over complex features containing smaller and less complex features. What this means is that integration testing doesn’t begin until unit testing of smaller features has been done. Once the smaller components are tested, then unit testing and integration testing of the more complex features can begin, including the expected normal, boundary, and exception cases in testing using the smaller components. As faults are revealed, either the smaller unit will be changed and retested or the complex feature being integrated will be reworked to accommodate the smaller features needs and system’s needs.

## **III.3 System Testing**

System testing is a type of black box testing that tests all the components together, seen as a single system to identify faults with respect to the scenarios from the overall requirements specifications. Entire system is tested as per the requirements.

During system testing, several activities are performed.

## **III.3.1 Functional testing:**

Test of functional requirements (from requirements specification). The goal is to select those tests that are relevant to the user and have a high probability of uncovering a failure.

**Event Write/Read Test:**

This test will use the event writer to write to the stream and then use the event reader to read from the stream. The test will then check what is read from the stream matches what was written. It will also record the time needed to fully perform the test.

**Event Reader Group/Transaction Writer Test:**

This test will use transaction writer to write events over multiple transaction. Then reader group will collectively read all the events in the stream. The test will then check to ensure all the events are what was written earlier by the transaction writer.

**Manage Transactions Test:**

This test will attempt to apply all possible actions on transactions. This includes committing, flushing, checking status, pinging and aborting. After each action is applied the test will check for the correct result on the transaction.

**Byte Writer/Reader Test:**

This test will write to the stream with byte writer only applying the essential bytes without any headers or encoding. Byte reader then will read from the stream. Once the reader is finished the output will be compared to the original input.

**State Synchronizer Test:**

This test will apply the state synchronizer to the system. The test will then check if the state is synchronized correctly over the processes.

**Table Synchronizer Test:**

This test will apply the table synchronizer to the system. Then the test will purposefully pause the client state and continue to use event writer to write events to the stream. After the events are written it will record the states of the client and server. Once recorded it will remove the pause from the client state. After giving multiple minutes, check to see if the client state is up to date with the server state.

**Stream Manager Test:**

This test will use the stream manager to create scopes, streams, write and readers. Then it will apply the correct functions to test if the newly created systems work correctly.

## **III.3.2 Performance Testing**

Performance tests check whether the nonfunctional requirements and additional design goals from the design document are satisfied. In stress testing, system is stressed beyond its specifications to check how and when it fails.

For performance testing we will be comparing the usage analytics of the rust client compared to the C# wrapped rust client. The exact metrics we will be recording are memory usage, storage usage, and time. There will be certain situations setup to compare the two programs. An example of one would be running the event writer. We will record the memory, storage and time taken to complete on the rust client and then on the C# wrapped client. After the metrics are recorded, we will compare to notice any inconsistent metrics that may cause problems for the end user.

## **III.3.3 User Acceptance Testing**

Acceptance testing and installation testing check the system against the project agreement. The purpose is to confirm that the system is ready for operational use. During acceptance test, end-users (customers) of the system compare the system to its initial requirements (if necessary) with help by the developers.

Once the system is considered to be complete by our team, we will deliver the product to the Pravega API team. We will ask them to try out the program and give them the list of essential features as well as all the extra features we were able to complete.

## **IV. Environmental Requirements**

Specify both the necessary and desired properties of the test environment. The specification should contain the physical characteristics of the facilities, including the hardware, communications and system software, the mode of usage (for example, stand-alone), and any other software or supplies needed to support the test. Identify special test tools needed.

For the test environment in C# we will be using NUnit to perform the tests[15]. The tests will be performed initially locally with the server running on the localhost. Later in the development process we will have a Pravega server running to test our client over the internet. For performance testing we will use a dummy server that returns the success of all the methods with metrics. We can also use that same dummy server to get the metrics for the C# code.

**V. Test Results**

**V.I Module Tests**

**[Client Factory Module]**

**Unit Tests**

**Default Constructor (ClientFactory)**

Aspect Being Tested: new ClientFactory(). Functionality

Expected Result: Client Factory is created with default settings (Pravega default ClientConfig, runtime generated tokio::runtime::Runtime) and contains a reference back to Rust generated memory where the data is stored.

Observed Result: Client Factory was created with default settings and non-null reference to its Rust counterpart.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Default Constructor time (ClientFactory)**

Aspect Being Tested: ClientFactory()’s time efficiency compared to rust. Functionality

Expected Result: From the average of 10 tests the time to create a default ClientFactory Object is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor with config (ClientFactory)**

Aspect Being Tested: new ClientFactory(ClientConfig). Functionality

Expected Result: Client Factory is created with inputted ClientConfig ( And runtime generated tokio::runtime::Runtime) and contains a reference back to Rust generated memory where the data is stored. Config inputted was set to null afterwards.

Observed Result: Client Factory was created with config and non-null reference to its Rust counterpart. Config inputted was set to null afterwards.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor with config time (ClientFactory)**

Aspect Being Tested: new ClientFactory(ClientConfig)’s time efficiency. Functionality

Expected Result: From the average of 10 tests the time to create a ClientFactory Object with an inputted ClientConfig is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor with config and runtime (ClientFactory)**

Aspect Being Tested: new ClientFactory(ClientConfig, TokioRuntime). Functionality

Expected Result: Client Factory is created with inputted ClientConfig and TokioRuntime and contains a reference back to Rust generated memory where the data is stored. Config inputted and Runtime inputted were set to null afterwards.

Observed Result: Client Factory was created with config and runtime and non-null reference to its Rust counterpart. Config inputted and Runtime inputted were set to null afterwards.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor with config and runtime time (ClientFactory)**

Aspect Being Tested: new ClientFactory(ClientConfig)’s time efficiency. Functionality

Expected Result: From the average of 10 tests the time to create a ClientFactory Object with an inputted ClientConfig and Runtime is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Runtime getter function (ClientFactory)**

Aspect Being Tested: ClientFactory.Runtime. Functionality

Expected Result: Runtime is able to be retrieved from a generated ClientFactory. Retrieved reference isn’t null.

Observed Result: Runtime was able to be retrieved from a generated ClientFactory. Retrieved reference wasn’t null.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Runtime getter function time (ClientFactory)**

Aspect Being Tested: ClientFactory.Runtime’s time efficiency. Functionality

Expected Result: From the average of 10 tests the time to retrieve a ClientFactory’s runtime is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Handle getter function (ClientFactory)**

Aspect Being Tested: ClientFactory.Handle. Functionality

Expected Result: Handle is able to be retrieved from a generated ClientFactory. Retrieved reference isn’t null.

Observed Result: Handle was able to be retrieved from a generated ClientFactory. Retrieved reference wasn’t null.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Handle getter function time (ClientFactory)**

Aspect Being Tested: ClientFactory.Handle’s time efficiency. Functionality

Expected Result: From the average of 10 tests the time to retrieve a ClientFactory’s handle is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Config getter function (ClientFactory)**

Aspect Being Tested: ClientFactory.Config. Functionality

Expected Result: ClientConfig is able to be retrieved from a generated ClientFactory. Retrieved reference isn’t null.

Observed Result: ClientConfig was able to be retrieved from a generated ClientFactory. Retrieved reference wasn’t null.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Config getter function time (ClientFactory)**

Aspect Being Tested: ClientFactory.Config’s time efficiency. Functionality

Expected Result: From the average of 10 tests the time to retrieve a ClientFactory’s config is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**ToAsync function (ClientFactory)**

Aspect Being Tested: ClientFactory.ToAsync(). Functionality

Expected Result: ToAsync() returns a ClientFactoryAsync object whose reference isn’t null.

Observed Result: ToAsync() returned a ClientFactoryAsync object whose reference isn’t null.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**ToAsync function time (ClientFactory)**

Aspect Being Tested: ClientFactory.ToAsync()’s time efficiency. Functionality

Expected Result: From the average of 10 tests the time to run ToAsync() is at least 0.85 as efficient as in Rust, that is (C# time / Rust time) > 0.85.

Observed Result: C# time < 0 nanoseconds, Rust time < 0 nanoseconds.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**[Byte Module]**

**[Synchronizer Module]**

**V.II Non-Module Tests**

**[Authentication Wrapper]**

**[Config Wrapper]**

**Default Constructor (ClientConfig)**

Aspect Being Tested: new ClientConfig(). Functionality

Expected Result: A new ClientConfig is created with Pravega’s default config settings. Reference of new object in C# is not null.

Observed Result: A new ClientConfig was created with Pravega’s default config settings. Reference of new object in C# was not null.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**MaxConnectionsInPool setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.MaxConnectionsInPool. Functionality

Test Cases: [0,10,Max unsigned integer]

Expected Result: The MaxConnectionsInPool is able to be set and retrieved through Rust dll calls for all test cases. Value remains the same after being set.

Observed Result: The MaxConnectionsInPool was able to be set and retrieved through Rust dll calls for all test cases. Value was observed to be the same after being set.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**MaxControllerConnections setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.MaxControllerConnections. Functionality

Test Cases: [0,10,Max unsigned integer]

Expected Result: The MaxControllerConnections is able to be set and retrieved through Rust dll calls for all test cases. Value remains the same after being set.

Observed Result: The MaxControllerConnections was able to be set and retrieved through Rust dll calls for all test cases. Value was observed to be the same after being set.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**RetryPolicy setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.RetryPolicy. Functionality

Expected Result: The RetryPolicy (type of RetryWithBackoff) is able to be set and retrieved through Rust dll calls. Value remains the same after being set.

Observed Result: The RetryPolicy was able to be set and retrieved through Rust dll calls. Value was observed to be the same after being set.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**ControllerUri setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.ControllerUri. Functionality

Expected Result: The ControllerUri (type of PravegaNodeUri) is able to be set and retrieved through Rust dll calls. Value remains the same after being set.

Observed Result: The ControllerUri was able to be set and retrieved through Rust dll calls. Value was observed to be the same after being set.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**TransactionTimeoutTest setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.TransactionTimeoutTime. Functionality

Test Cases: [0,10,Max unsigned long integer (64 bits)]

Expected Result: The TransactionTimeoutTime is able to be set and retrieved through Rust dll calls for all test cases. Value remains the same after being set.

Observed Result: The TransactionTimeoutTime was able to be set and retrieved through Rust dll calls for all test cases. Value was observed to be the same after being set.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Mock setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.Mock. Functionality

Expected Result: The Mock value is able to be set and retrieved through Rust dll calls. Value remains the same after being set.

Observed Result: The Mock value was set to false and then after true. Was able to be retrieved after and observed to not change.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**TlsEnabled setter and getter (ClientConfig)**

Aspect Being Tested: ClientConfig.IsTlsEnabled. Functionality

Expected Result: The IsTlsEnabled value is able to be set and retrieved through Rust dll calls. Value remains the same after being set.

Observed Result: The IsTlsEnabled value was set to false and then after true. Was able to be retrieved after and observed to not change.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**[Connection Pool Wrapper]**

**[Controller Client Wrapper]**

**[Retry Wrapper]**

**Default Constructor (RetryWithBackoff)**

Aspect Being Tested: new RetryWithBackoff (). Functionality

Expected Result: A new RetryWithBackoff is created with Pravega’s default retry policy settings. Reference of new object in C# is not null.

Observed Result: A new RetryWithBackoff was created with Pravega’s default retry policy settings through a .dll call. Reference of new object in C# was not null.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**[Shared Wrapper]**

**[Wire Protocol Wrapper]**

**[Utility]**

**Default Constructor (CustomCSharpString)**

Aspect Being Tested: new CustomCSharpString(). Functionality

Expected Result: A new CustomCSharpString is created that only contains “ “ in unmanaged memory.

Observed Result: A new CustomCSharpString is created that only contains “ “ in unmanaged memory that was able to be accessed.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor from string (CustomCSharpString)**

Aspect Being Tested: new CustomCSharpString(string). CustomCSharpString.NativeString (setter and getter from “string”) Functionality

Test Cases: [“test”,””]

Expected Result: A new CustomCSharpString is created that only contains “ “ when the inputted string is empty and contains the inputted string if it isn’t empty in unmanaged memory.

Observed Result: A new CustomCSharpString is created with the correct values inside depending on the test case.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor from a large string (CustomCSharpString)**

Aspect Being Tested: new CustomCSharpString(string). CustomCSharpString.NativeString (setter and getter from “string”) Functionality

Test Cases: a string containing 2^15 characters

Expected Result: A new CustomCSharpString is created that only contains the inputted large string and can be accessed without issue.

Observed Result: A new CustomCSharpString was created that only contains the inputted large string and could be accessed without issue.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor from rust string (8 bit) (CustomCSharpString)**

Aspect Being Tested: new CustomCSharpString(CustomRustString). CustomCSharpString.RustString (setter and getter from “CustomRustString”) Functionality

Test Cases: [“test”,””]

Expected Result: A new CustomCSharpString is created that only contains “ “ when the inputted string is empty and contains the inputted string if it isn’t empty in unmanaged memory. Converts from rust utf-8 to utf-16 successfully keeping in mind the limitations of the conversion.

Observed Result: A new CustomCSharpString was created that only contains “ “ when the inputted string was empty and contains the inputted string if it wasn’t empty in unmanaged memory.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Constructor from large rust string (8 bit) (CustomCSharpString)**

Aspect Being Tested: new CustomCSharpString(CustomRustString), CustomCSharpString.RustString (setter and getter from “CustomRustString”) Functionality

Test Cases: a string containing 2^15 characters

Expected Result: A new CustomCSharpString is created that only contains the inputted large string and can be accessed without issue.

Observed Result: A new CustomCSharpString was created that only contains the inputted large string and could be accessed without issue.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Clone (CustomCSharpString)**

Aspect Being Tested: CustomCSharpString.Clone() Functionality

Expected Result: A new CustomCSharpString is created from an existing one. The copy is a deep clone.

Observed Result: A new CustomCSharpString was created from an existing one. The copy’s references didn’t match and therefore was a deep clone.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Default Constructor (RustStructWrapper)**

Aspect Being Tested: new RustStructWrapper()

Expected Result: A new RustStructWrapper object is created with the object containing a reference to IntPtr.zero.

Observed Result: A new RustStructWrapper object is created with the object containing a reference to IntPtr.zero.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**IsEqual function (RustStructWrapper)**

Aspect Being Tested: RustStructWrapper.IsEqual(RustStructWrapper)

Expected Result: Given 2 generated RustStructWrapper objects that are the same, this function should return true if they are the same.

Observed Result: The function returned true when comparing the two objects that were the same.

Test Result: Pass

Test Case Requirements: Previously Specified Environmental Requirements

**Table of Test Results**

Created 2/2/2023. Generated from NUnit testing. Note that Duration of Test does not correspond to the time measured in the unit tests related to time efficiency. TestGroup indicates that multiple tests were ran on the same functionality and that there were different test cases.

|  |  |  |
| --- | --- | --- |
| **Test** | **Duration** | |
| RustStructIsEqualTest Passed | | 18 ms |
| RustStructDefaultConstructorTest Passed | | 7 ms |
| RetryWithBackoffDefaultConstructor Passed | | < 1 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringRustStringAndConstructorTest Passed (2) | | 13 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringNativeStringAndConstructorTest Passed (2) | | 17 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringFromCustomStringTest Passed (2) | | 17 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.CustomStringCapacityTest Passed Stale (2) | | 1 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ConfigTransactionTimeoutTest Passed (3) | | 1 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ConfigMaxControllerConnectionsTest Passed (3) | | 1 ms |
| Test Group: PravegaWrapperTestProject.PravegaCSharpTest.ConfigMaxConnectionsInPoolTest Passed (3) | | 1 ms |
| CustomStringFromBigStringTest Passed | | 489 ms |
| CustomStringFromBigRustStringTest Passed | | 617 ms |
| CustomStringDefaultConstructorTest Passed | | < 1 ms |
| CustomStringCloneTest Passed | | 1 ms |
| ConfigTlsEnabledTest Passed | | < 1 ms |
| ConfigRetryPolicyTest Passed | | 4 ms |
| ConfigMockTest Passed | | 1 ms |
| ConfigControllerUriTest Passed | | 38 ms |
| ConfigConstructor Passed | | < 1 ms |
| ClientFactoryToAsyncTest Passed | | 1 ms |
| ClientFactoryRuntimeTimeTest Passed | | 4 ms |
| ClientFactoryRuntimeTest Passed | | 1 ms |
| ClientFactoryHandleTimeTest Passed | | 3 ms |
| ClientFactoryHandleTest Passed | | 1 ms |
| ClientFactoryDefaultConstructorTimeTest Passed | | 17 ms |
| ClientFactoryDefaultConstructorTest Passed | | 14 ms |
| ClientFactoryConfigTimeTest Passed | | 3 ms |
| ClientFactoryConfigTest Passed | | 1 ms |
| ClientFactoryConfigRuntimeConstructorTimeTest Passed | | 4 ms |
| ClientFactoryConfigRuntimeConstructorTest Passed | | 3 ms |
| ClientFactoryClientConfigConstructorTimeTest Passed | | 3 ms |
| ClientFactoryClientConfigConstructorTest Passed | | 5 ms |
| ClientFactoryAsyncTimeTest Passed | | 16 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.

**Final Prototype**

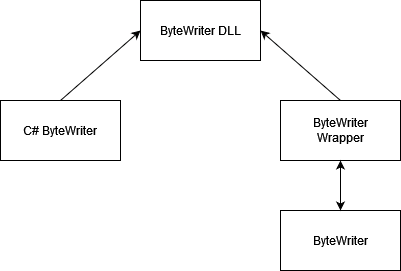
1. **Description**

Our final prototype utilizes the Foreign Function Interface (FFI) to convert Rust code into C# libraries. FFI allows the code to be converted down to the C level and then be rebuilt up to C# code. Once the code has been converted, it is then exported to a DLL file, which is a dynamic link library that contains the converted code. Once all of the rust code is correctly converted it allow the user’s to access the Pravega API with C# code.

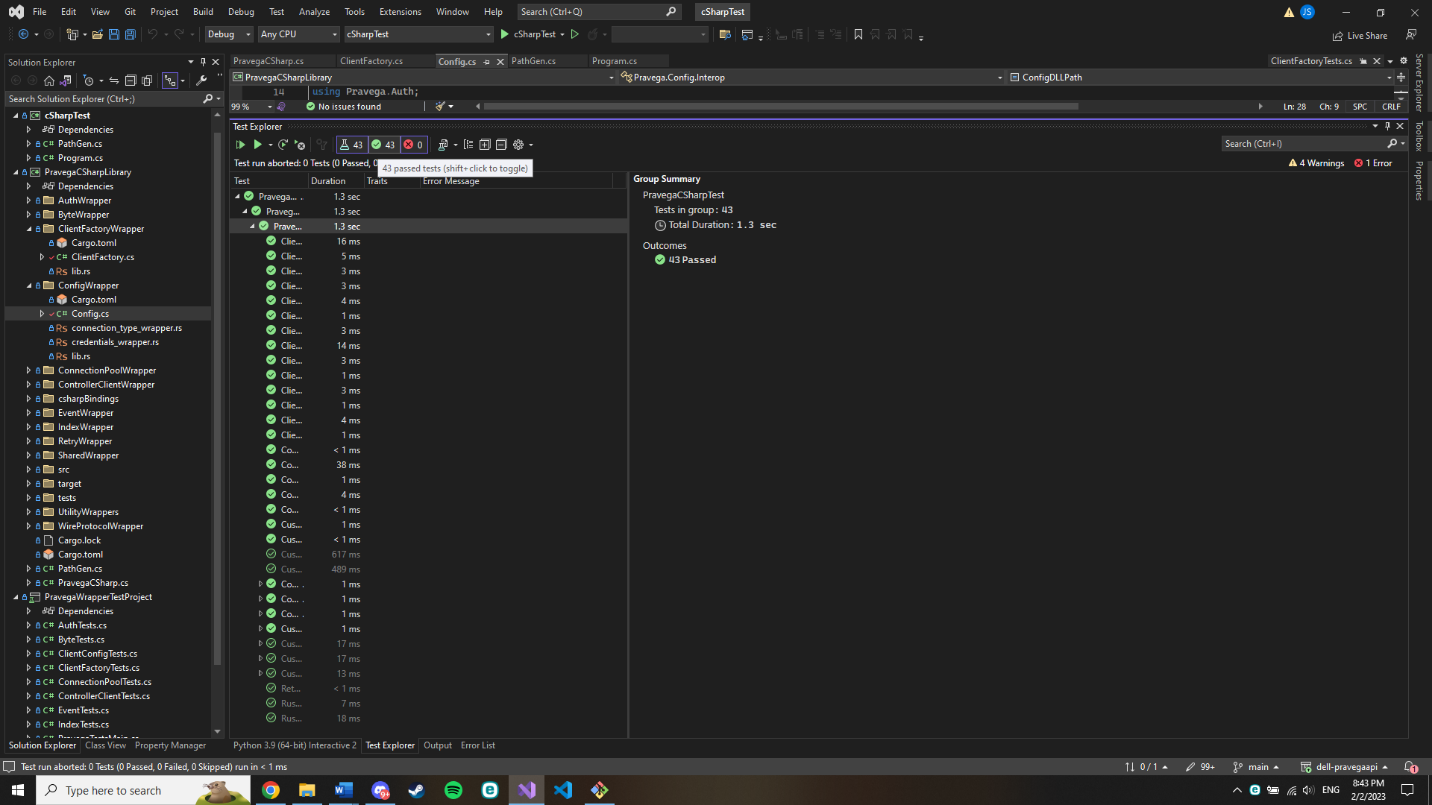
An important part of the prototype is creating Rust wrappers so the C# code works correctly. The goal is to have the memory management of objects to be handled in the Rust code, not by C#. If we let C# handle the memory it’s garbage collector could decide to remove the memory when it still is in use. To achieve this design, after you create the object in Rust you have a pointer to the memory allocated and return that to C#. The plan of the prototype is for this to be applied to all modules of the Pravega API by end of semester. These wrappers are then converted to a DLL as mentioned earlier and wrapped up into a C# package for developers to use.

Another crucial aspect of our Prototype is the way we engage with Rust's asynchronous system. The Pravega Rust codebase uses Tokio which provides the users with a asynchronous runtime. We designed our system to store the runtime, this allows the C# developer to run asynchronous functions will then take the runtime in Rust and use it in the Rust wrapper functions.

Below is an example of ByteWriter being wrapped.



Below is the output of our current C# wrapper tests.



Client Factory Wrapper Functions

* **new ClientFactory ();**
  + Creates a new default instance for ClientFactory using the default Pravega configuration settings and a generated runtime.
* **new(ClientConfig factoryConfig);**
  + Creates a new instance of ClientFactory using the inputted ClientConfig settings and a generated runtime. Sets ClientConfig to null afterwards.
* **new(ClientConfig factoryConfig, TokioRuntime factoryRuntime);**
* Creates a new instance of ClientFactory using the inputted ClientConfig settings and the inputted runtime. Sets ClientConfig and Runtime to null afterwards
* **TokioRuntime ClientFactory.Runtime**
  + Getter for the ClientFactory object’s TokioRuntime
* **TokioHandle ClientFactory.Handle**
  + Getter for the ClientFactory object’s TokioHandle
* **ClientConfig ClientFactory.Config**
  + Getter for the ClientFactory object’s configuration settings
* **ClientFactoryAsync ClientFactory.ToAsync()**
  + Returns the ClientFactory object’s async counterpart

Creates a Async Client Factory

**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.

**Project Delivery Status**

We are hoping to finish our project and deliver it to our client by May 2. Currently the project is accessed by pulling from the GitHub hosting it, and then launching the C# project in an IDE, such as Visual Studio 2022. Eventually, the project will be compiled into a C# library. This and the Rust .dll files needed can then be simply added to any C# project. It is the hope of us and our mentors that this project will be added as a standard C# library from Microsoft, and that it can be used in a C# project by simply adding it as a reference.

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.

**Future Work**

Due to feedback received from our mentors, we had to rethink our strategy in wrapping Pravega components. We had originally planned on creating custom wrappers for every module that a user may interact with, but our clients suggested simply storing the object without having C# understand it so that it can be passed into rust code. This has caused us to be slightly behind our original goal of having ClientFactory fully wrapped in the first semester. As such, we had to slightly update our plans for future work. We had made decent progress on wrapping ClientFactory in our original method, but now we need to research the best method for passing raw data into C#. Our current research shows us that working with pointers will be a good start, provided both Rust and C# recognize the memory as in use, which it seems like it should. Following that we will update our ClientFactory code and finish it completely. Then we will start on the other modules that ClientFactory spawns and implement those until we are satisfied with the functionality of our code base.

**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

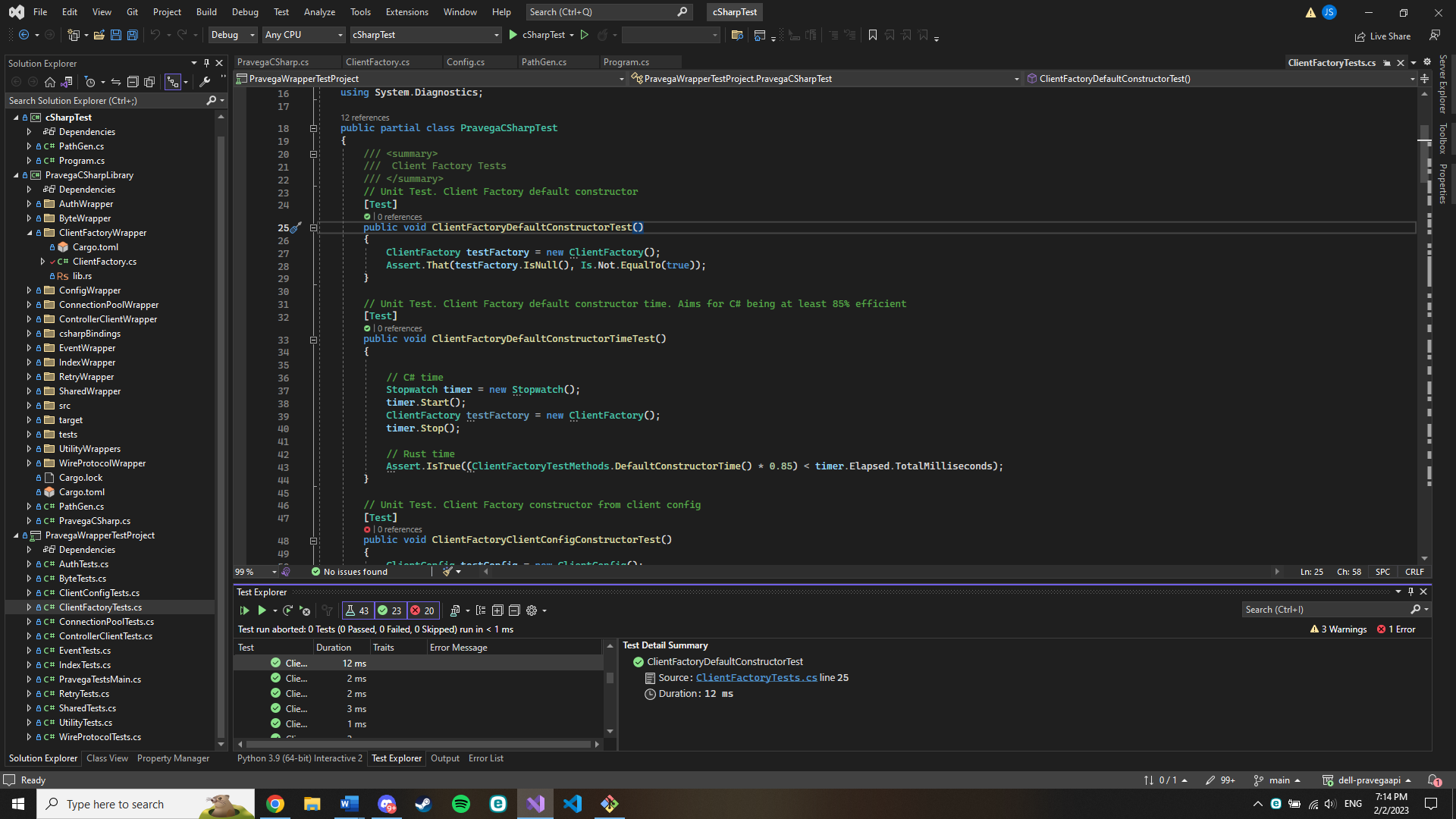
Results:

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#

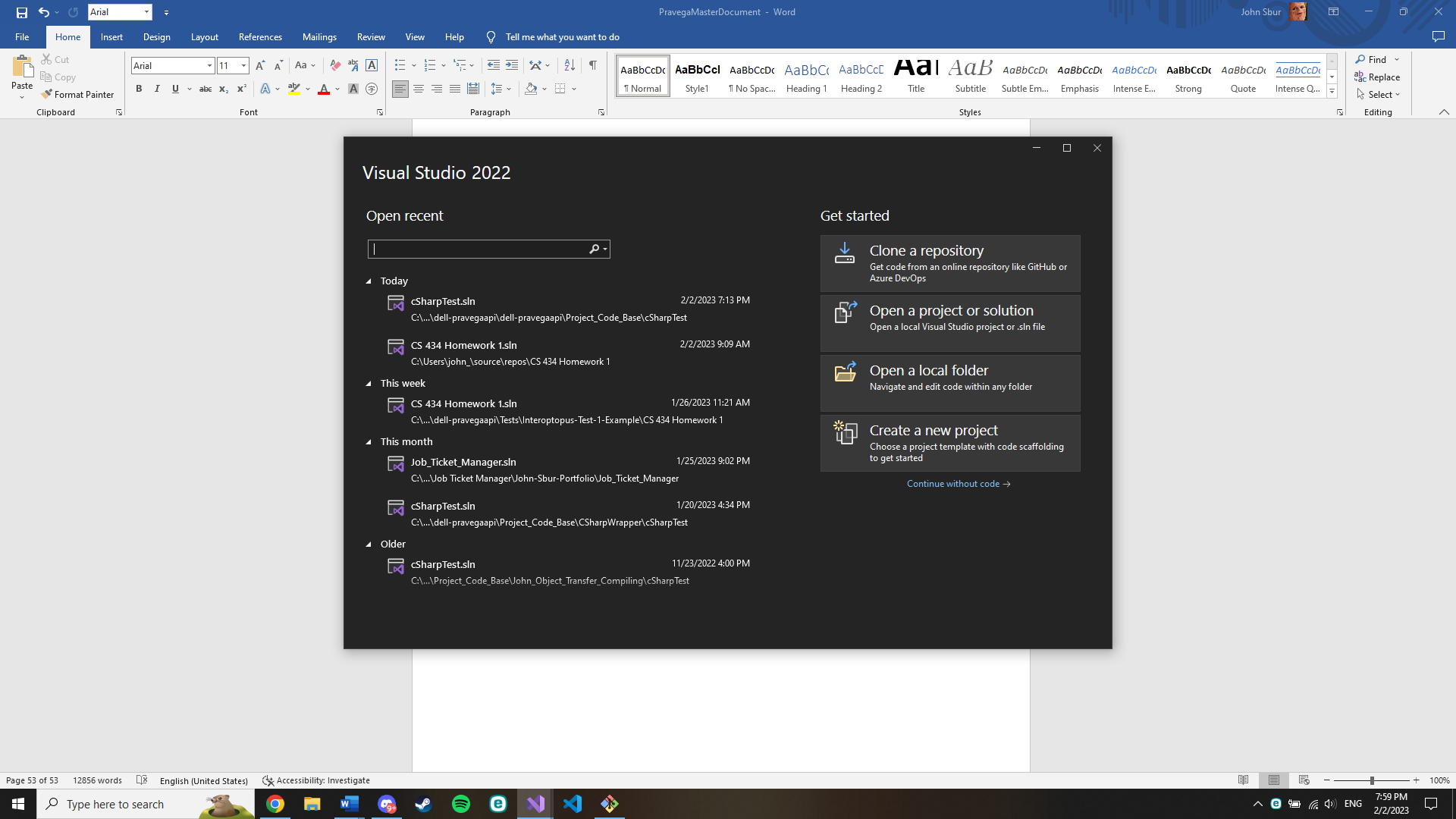
Test Result: Pass

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.