Pravega Client Library for .NET

Sponsored by: Dell Technologies

Icon

Description automatically generated

**By: The Vegateers**

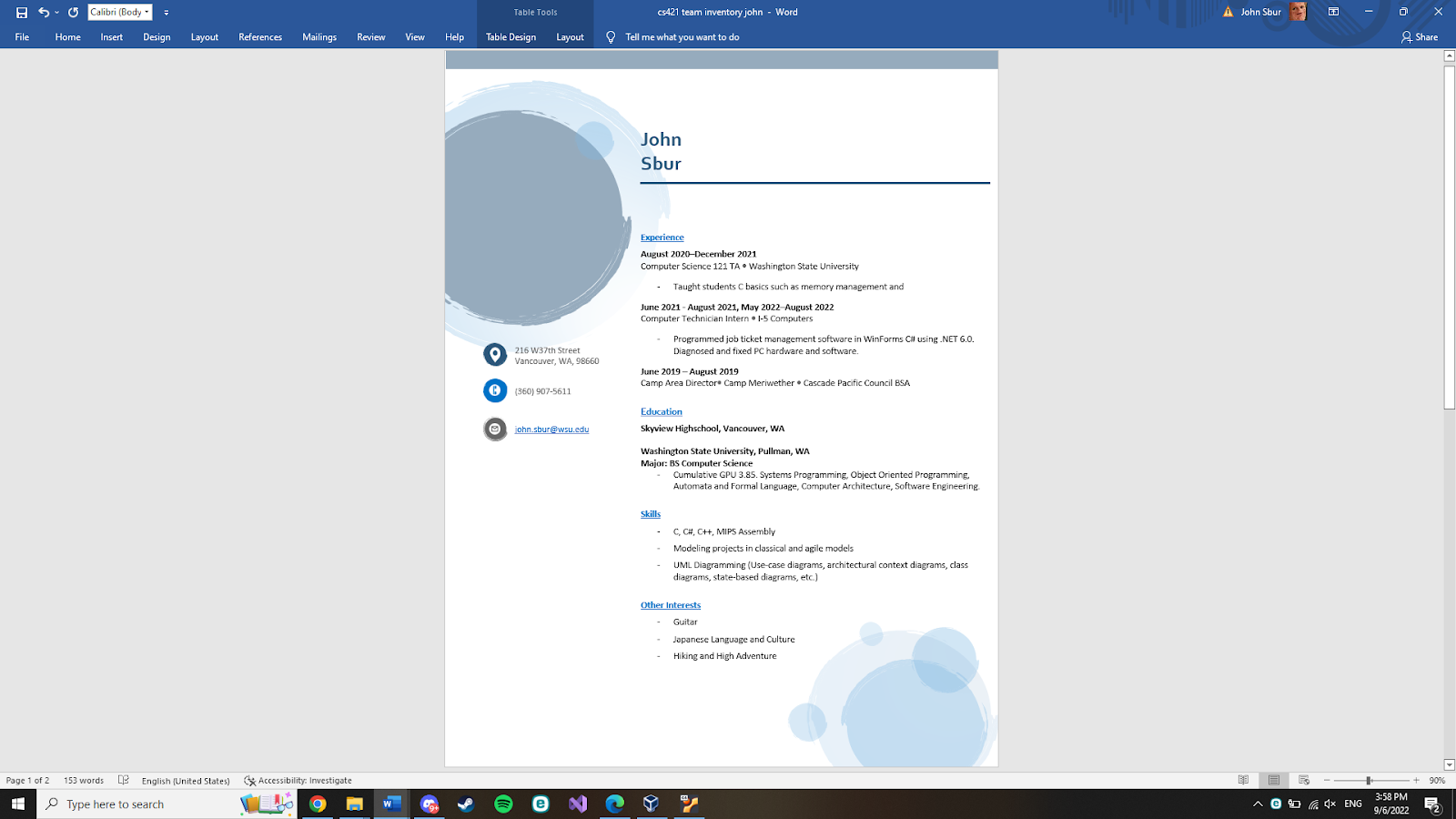
Sbur, John

Lopez, Samuel

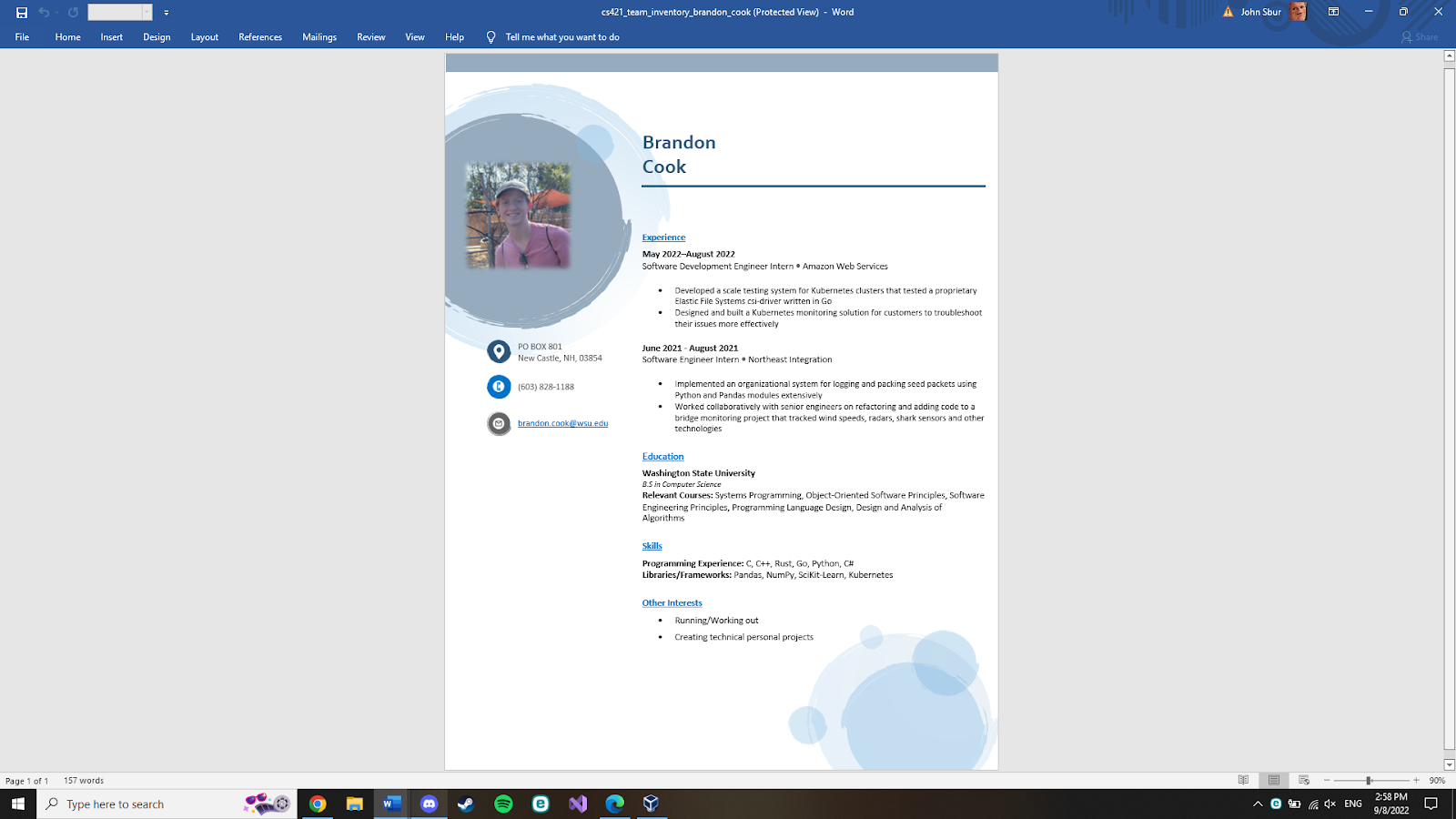
Cook, Brandon

**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 |
| Index | 26 |
| Client Factory | 29 |
| Sync | 30 |
| Error | 33 |
| 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 |
| Alpha Prototype Report | 44 |
| Alpha Prototype Description | 44 |
| Alpha Prototype Demonstration | 45 |
| Future Work | 50 |
| Glossary | 50 |
| References | 50 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |





Project Description

**Project Description**

1. **Introduction**

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

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

1. **Background and Related Work**

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

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

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

1. **Project Overview**

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

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

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

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

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

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

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

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

**Requirements and Specifications**

**I.\_Introduction**

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

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

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

**II. System Requirements Specifications**

**II.1 Use Cases**

Diagram

Description automatically generated

**Upload/Receive Data:**

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

Relevant Functional Requirements: All

**Read Bytes:**

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

Relevant Functional Requirements: Byte Reader

**Write Bytes:**

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

Relevant Functional Requirements: Byte Writer

**Read Events:**

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

Relevant Functional Requirements: Event Reader, Reader Group

**Write Events:**

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

Relevant Functional Requirements: Event Writer, Reader Group

**Manage Transactions:**

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

Relevant Functional Requirements: Managing Transactions

**Manage Streams:**

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

Relevant Functional Requirements: Stream Manager Client

**Synchronize State**

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

Relevant Functional Requirements: State Synchronizer

**Use Key Value Tables:**

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

Relevant Functional Requirements: Managing Transactions

**Synchronize Table:**

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

Relevant Functional Requirements: Table Synchronizer

**II.2. Functional Requirements**

**II.2.1. Event Stream**

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

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

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

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

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

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

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

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

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

**II.2.2. Transactions**

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

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

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

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

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

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

**II.2.3. Byte Client**

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

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

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

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

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

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

**II.2.4. Synchronizer**

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

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

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

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

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

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

**II.2.5. Key Value Tables**

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

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

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

**II.2.6. Stream Manager**

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

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

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

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

**[Language Specification]**

The languages we 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.

**Project 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 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].

**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 to be converted are byte, event, index, sync, client\_factory, and error. Each of these modules have their own complex data structures that are taken into consideration while wrapping as these data structures or the functionality of them should be used once converted to C# from Rust. For example, in index, the IndexReader and IndexWriter are located in the module and are used to read and write to locations in a stream. That functionality needs to be accounted for when wrapped.

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#. The FFI we have selected for this project is Interoptopus, as it is well documented and fulfills the functionalities we require. 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.

**III. Architecture Design**

**III.I. Overview**

A picture containing Excel

Description automatically generated

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. 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 that will be 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.

**III.II. Sub System Decomposition**

**[Byte]**

**Reader**

This subsystem is responsible for reading the byte code correctly. It has an algorithm that prefetches data from the server and caches it into memory. For creating a ByteReader instance you scope the segments and create an async reader. The ByteReader allows the user to read from the stream. You can create a new reader, asynchronously read and seek.

**Services Provided:**

**Name**: Reader

-Provided to: C# Developer

-Description: Reads from the byte stream.

**Writer**

This subsystem is responsible for writing the byte stream correctly. The system will only let you write 8 MiB at a time. Is able to write asynchronously. The ByteWrtiter allows the user to write to the stream with bytes. You can create a new writer asynchronous read.

**Services Provided:**

**Name**: Writer

-Provided to: C# Developer

-Description: Writes to the byte stream

Diagram

Description automatically generated

**[Event]**

Diagram, engineering drawing

Description automatically generated

*Event subsystem decomposition*

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 Readerstate 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.

**Services Provided:**

**Name:** Allow user to read from Pravega stream

- Provided to: C# Developer

- Description: Allows the C# user to read events from a Pravega stream by spawning a SegmentSlice.

**Services Required:**

Meta Provided by **ReaderState** and **ReaderGroupState**

Created in a program by a **ReaderGroup**

**Uses a SegmentSlice** to read data from a stream

**ReaderState**

The ReaderState is a subsystem that is used to store meta data about an EventReader. For example, it stores whether a reader is offline or not.

It has the functions addSliceReleaseReciever which adds a receiver to inform a reader when a SegmeentSlice is returned, waitForSegmentSliceReturns which waits until a SegmentSlice is returned, removeSegment which removes a segment from meta data, addSlices which adds SegmentSlices to the meta data, addStopReadingtx which stores a sender used to stop a read task, stopReading and stopReadingAll which stop a single and all background reads respectively.

**Services Provided:**

**Name:** Allow use of metadata

- Provided to: EventReader

- Description: Stores the metadata of an EventReader

**Services Required:**

Multiple features from **pravegaClientShared**

**SegmentSlice**

The SegmentSlice is a subsystem that is used to read events from a Pravega segment and acts like an iterator.

It has the functions getSegmentData which gets data from the Segment store from a given start offset, getStartingOffset which gives the starting offset, getSegment which gives the associated segment, extractEvent which gets the next event from the SegmentStore, and readHeader which reads the header and gives the size of the event.

**Services Provided:**

**Name:** Allow EventReader to give segments.

- Provided to: EventReader

- Description: Allows EventReader to give segments to the user.

**Services Required:**

Meta Provided by **SliceMetaData**

Provides event data via **Event**

**Event**

Event is a fairly simple subsystem. It simply stores the data of an event and the offset.

**Services Provided:**

**Name:** Allow **SegmentSlice** to provide meta data

- Provided to: SegmentSlice

- Description: Allows SegmentSlice to give users event data.

**SliceMetaData**

SliceMetaData stores the meta data for a SegmentSlice. For example, it contains the start offset and whether there is data present.

It has the Functions isEmpty which checks to see if a slice has partial data, hasEvents which checks that the slice has events to be read, and copyMeta to copy itself.

**Services Provided:**

**Name:** Store Meta Data

- Provided to: SegmentSlice

- Description: Stores the metadata for a SegmentSlice

**Services Required:**

Uses a **SegmentDataBuffer** to store the offset and byte array.

**SegmentDataBuffer**

The SegmentDataBuffer is used to store the offset and byte array of a segment. This is than stored in SegmentMetaData.

It has the functions split and splitTo, which split the data buffer, getI32 which gives the first integer in the buffer and advances by 4 bytes, advance which advances the internal cursor of the buffer, and empty which produces an empty SegmaneDataBuffer.

**Services Provided:**

**Name:** Store segment data

- Provided to: SegmentMetaData

- Description: Allows SegmentMetaData to store the bytes of a given segment.

**ReaderGroup**

The ReaderGroup is a group of events that can be used to read stream data. It is responsible for creating EventReaders, as every EventReader needs to be part of a ReaderGroup.

It has the functions create which creates a ReaderGroup, delete which deletes a ReaderGroup, create\_reader which is used to create EventReaders, listReaders which lists all readers in a group, and reader\_offline which removes a reader from the ReaderGroup.

**Services Provided:**

**Name:** Allow user to read from Pravega stream

- Provided to: C# Developer

- Description: Allows the C# user to read events from a Pravega stream by spawning an EventReader.

**Services Required:**

Meta Provided **ReaderGroupState**

Config specified by **ReaderGroupConfig**

Config created by **ReaderGroupConfigBuilder**

**ReaderGroupConfigBuilder**

The ReaderGroupConfigBuilder is used to build ReaderGroupConfigs for a ReaderGroup. It controls streams and sets the refresh time.

It has the functions setGroupRefreshTime, addStream and readFromHeadofStream which add a stream to the ReaderGroup and read from the start of the stream, ReadFromTailOfStream which reads from the tail of a stream, and readFromStream which reads from a specified streamcut.

**Services Provided:**

**Name:** Create Config

- Provided to: ReaderGroup

- Description: Creates a ReaderGroupConfig for a Readergroup.

**Services Required:**

Creates a ReaderGroupConfig with a **StreamCutVersioned**

**ReaderGroupConfig, ReaderGroupConfigVersioned, ReaderGroupConfigV1**

These three subsystems work together to provide config for a ReaderGroup. They work in a nested fashion with ReaderGroupConfig holding a ReaderGroupConfigVersioned which in turn holds a ReaderGroupConfigV1.

ReaderGroupConfig has the functions toBytes and fromBytes to serialize and deserialize to bytes, get\_streams to obtain the streams store inside, and get\_start\_stream\_cuts to obtain streams and the start streamcut.. ReaderGroupConfigVersioned has the functions toBytes and fromBytes. ReaderGroupConfigV1 has the function start\_from\_stream\_cuts.

**Services Provided:**

**Name:** Config

- Provided to: ReaderGroup

- Description: Provides the config for a ReaderGroup.

**Services Required:**

Stream cuts provided via **StreamCutVersioned**

**StreamCutVersioned and StreamCutV1**

The StreamCutVersioned and StreamCutV1 systems are used to hold data about all versions of a Streamcut struct and the segment/offset pairs.

**Services Provided:**

**Name:** StreamCut

- Provided to: ReadGroup

- Description: Allows the ReaderConfig to use StreamCut

**ReaderGroupState**

The ReaderGroupState encapsulates all readers states and has a synchronizer that allows the user to read or write the internal reader group state stored on the server.

It has the functions addReader which adds a reader to the group state, check\_online, getOnlineReaders which returns active readers, getReaderPositions which gives the position of a given reader,updateReaderPositions which updates a readers position, remove\_reader\_default which removes a reader and puts marks its segments as unassigned,removeReader which removes a specific reader, computeSegmentsToAcquireorRelease, getSegments which gives a list of all segments, assignReaderToSegment which assigns a segment to a given reader, getSegmentsForReader which gets the segments for a specified reader, releaseSegment which releases a currently assigned segment, segmentCompleted which removes the completed segments, getReaderOwnedSegmentsFromTable, getUnassignedSegmentsFromTable, getFutureSegmentsFromTable, and CheckReaderOnline.

**Services Provided:**

**Name:** ReaderGroupState

- Provided to: ReaderGroup, EventReader

- Description: Encapsulates all reader states in a ReaderGroup

**Services Required**

-Various functions use the **Offset** subsystem

**Offset**

The Offset has an i64 that the client has read to.

**Services Provided:**

**Name:** Various

- Provided to: ReaderGroupState

- Description: Proved functional utility in various ways

**TransactionalEventWriter**

The TransactionalEventWrites is used to write events to a stream transactionally. It is created by a ClientFactory and spawns transactions for the user to interact with. The transactions write events that can be confirmed or aborted in the future by the user.

It has the functions begin which begins a transaction by sending a request to a controller, and get\_txn which returns a transaction based on a given ID.

**Services Provided:**

**Name:** Create Transactions

- Provided to: C# developer

- Description: Allows the user to develop transactions.

**Services Required:**

Uses the Transaction subsystem to give user transaction writing capabilities.

**Transaction**

This subsystem is used to create transactional events. After they are written, the user is free to commit the events or abandon them as needed.

It has the functions txn\_id which gets the transactions ID, stream which gets the stream the transaction is based on, writeEvent which writes the actual bytes of an event, commit which commits a transaction, and abort which aborts a transaction.

**Services Provided:**

**Name:** Write transaction

- Provided to: user, TransactionalEventWriter

- Description: allow TransactionalEventWriter to provide end users with a way to write transactions.

**Services Required:**

Meta data provided by **TransactionInfo**

**TransacitonInfo**

This subsystem simply holds the metadata for a transaction.

**Pinger and PingerEvent**

Pinger is a subsystem that pings transactions it a background task. PingerEvent is an enum that allows for connection between a Pinger and Pinger handler.

Pinger has the functions startPing which begins pinging, and pingInterval which changes the interval of pings

**Services Provided:**

**Name:** Ping

- Provided to: C# Developer

- Description: Allows the user to ping transactions.

**Services Required:**

* **PingerHandler can be used to add or remove transactions.**

**PingerHandler**

Pinger handler is a handler that allows for adding and removing a transaction from a Pingers list of transactions.

**Services Provided:**

**Name:** Add/Remove

- Provided to: Pinger

- Description: Add and Remove transactions from a Pinger

**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.

**Services Provided:**

**Name:** Write Events

- Provided to: C# Developer

- Description: Allows a user to write events to a Pravega stream.

Graphical user interface, diagram

Description automatically generated with medium confidence**[Index]**

*Subsystem overview*

A screenshot of a computer

Description automatically generated*Subsection decomposition shown above*

**Mod**

Mod is a subsection that mainly contains subsystem header content like library inclusions and class declarations. In addition, however, is the inclusion of IndexRecord. IndexRecord is designed to hold a fixed sized chunk of data containing user data and the number of fields that user has defined. These are written and read from the Pravega servers as information tied to a user.

Services Provided:

-Name: User field management

-Provided to: C# developers

-Description: Allows users to classify and manage their fields on a Pravega data stream.

**IndexReader**

This subsection contains a reader class that reads IndexRecords from a Pravega data stream. The streams are single byte streams and contains previously written user data. Data can be removed at an offset, at the beginning, or conditionally from a stream as was designed in the original Pravega library.

**Services Provided:**

**Name**: Stream Index Reader

-Provided to: C# Developer

-Description: Allows reading of user data fields based on index inside of a Pravega stream.

**IndexWriter**

This subsection serves as an essential opposite to reader as a user is able to write user data and their fields to an IndexRecord through this class. Data can be inserted at an offset, at the beginning, or conditionally into a stream, reflecting the content in the original Pravega library.

**Services Provided:**

**Name**: Stream Index Writer

-Provided to: C# Developer

-Description: Allows writing of user data fields based on index inside of a Pravega stream.

**[ClientFactory]**

Diagram

Description automatically generated

**ClientFactory**

This system is a factory to create components for the Pravega client.

It also holds the connection pool for the readers and writers created.

For interfacing with the ClientFactory you can create a new instance. You can configure the client, create/delete reader groups and create byte writers/readers.

**Services Provided:**

**Name**: ClientFactory

-Provided to: C# Developer

-Description: Manages client instances

**ClientFactoryAsync**

This system is a factory to create asynchronous components for the Pravega client.

The system creates a tokio runtime that is needed to run the asynchronous tasks. It also holds the connection pool for the readers and writers created.

For interfacing with the ClientFactory you can create a new asynchronous instance. You can configure the client, create/delete reader groups and create byte writers/readers for asynchronous uses.

**Services Provided:**

**Name**: ClientFactoryAsync

-Provided to: C# Developer

-Description: Manages client asynchronous instances

**[Sync]**

*Subsystem Overview*A picture containing diagram

Description automatically generated*Synchronizer.cs, a file containing the synchronizer and update control, shown above*

**Synchronizer**

*Synchronizer.cs, a file containing the synchronizer and update control, shown above*

**Synchronizer**

*Synchronizer.cs, a file containing the synchronizer and update cfile containing the synchronizer and update control, shown above*

**Synchronizer**

Diagram

Description automatically generated with medium confidence*Synchronizer.cs, a file containing the synchronizer and update control, shown above*

**Synchronizer**

The synchronizer subsystem is responsible for handling data being streamed between multiple processes at the same time. It handles version control of data objects between processes and updates streams as data is removed and added between processes.

These classes, interfaces, and methods are pulled directly from the Pravega API existing and converted to C# terminology or use methods in C# that would act the same way. For instance, while Option exists in Rust as a type of data, it doesn’t in C#. An option means that the data contains a variable of a certain data type or it doesn’t. In C# terms, this is very similar to using T?, where T is a data type. This represents a variable of datatype T or null. Another similarity was traits in Rust acting similarly to interfaces in C# where objects that implement a trait or interface need to implement the functions or properties they require. While both of these don’t act exactly like each other, they are fairly close for the purposes of this project that we will not lose functionality.

**Services Provided:**

**Name**: Multiple Process Data Stream Management

-Provided to: C# Developer

-Description: With key version control, this service lets a developer manage mapped data over multiple processes without losing accuracy. Users are able to insert and remove data from a map and have that change resonate between all other processes using that map.

**Name**: Conditional Version Control

-Provided to: C# Developer

-Description: Allows a C# developer to manage when processes using the same table update and if they update.

*Table.cs, a file containing the storage method for a synchronizer and, shown on previous page.*

Graphical user interface, application, table, Excel

Description automatically generated**Table**

The table subsection is responsible for holding all process stream segments in one place to be managed. It is able to create, modify, and delete process stream segments upon request and do so conditionally if needed.

These algorithms are derived directly from the Rust Pravega library in C# format and styling. Much like other modules, objects that exist in Rust but not in C# are replaced with equivalents in this implementation.

**Services Provided:**

**Name**: Stream management across different processes under one object

-Provided to: C# Developer

-Description: Allows creation, deletion, and modification of streams across multiple processes running at the same time. Works as a part of a synchronizer to update processes as changed occur.

**[Error]**

Error is a simple system comprised of a single .rs file. As such, the most logical way to implement it is to have a simple error.cs file.

**Subsystem Breakdown**

**Error**

Error allows for detection and resolution of errors conditional check, internal failure, and invalid input.

**IV. Data Design**

**Byte Module**

reader.rs

Struct ByteReader:

* This structure allows the system to read raw bytes from a segment. It has similar features to Read and Seek traits in the standard library but can do them asyncrhonous. The reader can prefetch the data and store it in memory. The functions implemented for the ByteReader struct are new, read, current\_head, current\_tail, current\_offset, available, seek and recreate\_reader\_wrapper.

Writer.rs

Struct ByteWriter:

* This structure allows the system to write raw bytes to a segment. This writer does not modify any of the bytes. The only way to read bytes written from ByteWriter is with ByteReader. The Writer can only right with 8 MiB at a time. The system allows parallelism by using conditional append. The writer may also retry to writer if connection failures occur. The functions implemented for ByteWriter struct are new, write, flush, seal, truncate\_data\_before, current\_offset, seek\_to\_tail, reset, write\_internal.

**Event Module**

Reader.rs

Enum EventReaderError:

* Used to indicate a reader group state error

Struct 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 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.

Struct ReaderState:

* This struct holds reader metadata

Struct Event:

* This represents an event that was read from a segment and the offset.

Struct SegmentSlice:

* This represents a segment slice and can be used to read events from a Pravega stream as an iterator.

Struct SliceMetaData:

* This represents the meta data of the segment slice.

Struct SegmentDataBuffer:

* Used to track offset and byte array

Reader\_group.rs

Struct ReaderGroup:

* This is the structure that is used to store and create eventreaders. Every EventReader must belong to a ReaderGroup. It has the functions create,delete, and reader\_offline.

Struct ReaderGroupConfig:

* This specifies the reader group config.

Struct ReaderGroupConfigBuilder:

* Used to build Reader group configs.

Enum ReaderGroupConfigVersioned:

* This contains all versions of the position struct.

Struct ReaderGroupConfigV1:

* This struct holds the max delay that the readers return offsets on their assigned segments

Enum StreamCutVersioned:

* This holds all of the versions of the StreamCut struct

Struct StreamCutv1:

* A set of segment and offset pairs for a stream

Enum SerdeError:

* Error enum for the readergroup file

reader\_group\_state.rs

Struct ReaderGroupState:

* This encapsulates all of the reader states. It contains a synchronizer used to read and write the internal reader group state.

Struct Offset:

* Represents the offset that the client has read to and handle the result to the caller.

Transactionsl\_writer.rs

Struct TransactionalEventWriter:

* This struct allows the user to write to a stream transactionally. It has methods begin and get\_txn

Struct TransactionInfo:

* This is used to store the metadata of the Transaction.

Struct Transaction:

* This is created by a transactioneventwriter to write and commit/abort a transaction to Pravega.

Enum PingerEvent:

* This is used to store information about a pinger

Struct Pinger:

* This is used to ping transactions periodically.

Struct PingerHandle:

* A wrapper used to communicate with a pinger

Enum TransactoinalEventWriterError:

* Used for errors on pinger and the streamcontroller.

Enum TransactionError:

* Used for errors with the Transaction struct

writer.rs

Struct EventWriter:

* This is the struct used to write events.It has functions write\_event, write\_event\_by\_routing\_key,writer\_event\_internal, flush, and clear\_initial\_complete\_events

**Index Module**

Reader.rs

Struct IndexReader:

* A structure designed to read data from single-segment streams. Assumes stream is fixed size. When using this structure, the basic “read()” function implemented gets and returns a slice of data from the stream as an iterator. Values can also be searched via their offset from a point in the stream read.

Enum IndexReaderError:

* An Enum that binds error messages to errors. Errors include invalid offset, field not found, and internal error.

Writer.rs

Struct IndexWriter:

* A structure designed to write a fixed size data segment to a stream. Before it writes to a stream, it takes an inputted raw byte array with data and data Fields, hashes each data by field, then serializes it (turns the data into a raw byte array) and sends it to a stream. When data is inputted, it is always appended onto the end of the stream it is writing to. In addition, data inputted can be limited to a certain amount through truncation giving a user control of exactly how much data is uploaded. Whether data is written or not can also be made to follow a condition.

Enum IndexWriterError:

* A, Enum that binds errors to error messages. Errors accounted for are invalid data, invalid fields, invalid condition, and internal.

**Client Factory Module**

Client\_factory.rs

Struct ClientFactory:

* This structure holds only two variables runtime and client\_factory\_async. The runtime is used to run asynchronous tasks.

Struct ClientFactoryAsync:

* This structure is designed to be used when tasks are running asynchronous. It holds a connection\_pool, controller\_client, config and runtime\_handle.

**Synchronizer Module**

Synchronizer.rs

Struct Synchronizer:

* This structure’s goal is to have a map that maintains data across multiple processes running at the same time and keeps each process up to date while doing do. It works by comparing the process Synchronizer’s state to the server’s state and updating the process Synchronizer’s state if it doesn’t match the server’s. The implication is that the map can be updated by any process by using the methods Insert and Remove. In addition, a process can update it with these methods, but won’t be recognized by another process until the method “fetchUpdates()” is called by that process.
* The in-memory data of a synchronizer is stored in a HashMap that uses two keys to identify a value. The HashMap itself contains a HashMap nested as the value part of the (key,value) pair. Implemented as HashMap<String, Hashmap<Key, Value>>.
* The nested HashMap is implemented for keeping track of versions and is handled by a Synchronizer through its in-memory-map-version which is that Synchronizer’s own version of the HashMap. When updated, a Synchronizer can check and see if its version is up to date and conditionally update if it isn’t.

Struct Key:

* Since the Synchronizer uses a HashMap to store data, a key is important to distinguish location. In addition, this key comes with a version attached to it. When a value in a HashMap using this key is updated, the version is as well, which can be checked by other processes to see if their version matches the version the key has. A notable behavior case is that if the version of the key is the minimum value for a 64-bit integer, it will update when checked no matter the case.

Struct InternalKey:

* Only contains a string and is used for parsing a key sent by the server.

Struct Value:

* This structure maintains data given to the table. It contains a string that tells methods using the structure the datatype it contains as well as a vector of bytes used to represent that datatype.

Struct Update:

* Much the same as the Synchronizer class, containing a map, map version, insert, and remove, though the functionality is different as it is one of the first steps in applying changes to a synchronizer. When an update is done in this class, it is sent to the Pravega server and after being stored, the change is then applied to the synchronizer.

Struct Insert:

* This struct focuses on updating or adding to the server side’s HashMap using the server’s key. The type of data needs to be specified as otherwise it is inserting a vector of bytes with no way to know what datatype that vector is trying to represent.

Struct Remove:

* This struct focuses on removing a value from the HashMap based on the server’s key. Unlike Insert, it doesn’t specify datatype.

Trait ValueClone:

* Gives a value with this trait the ability to be cloned

Trait ValueSerialize:

* Gives a value the ability to be turned from formatted data into raw bytes. Notably, methods exist for serializing, turning into raw bytes, and deserializing, turning raw bytes into formatted data, within this scope.

Enum SynchronizeError:

* An Enum used to bind different errors to different messages. These include update errors, tombstone errors, and precondition errors.

Struct Tombstone:

* Tombstones are in a sense a status of data. A data that is tombstoned means it is dead and ready to be removed, but to save time it isn’t removed. It’s similar to how a process running on a computer is marked as a Zombie when completed before being removed later or how data blocks in a file system are marked as free while the original contents inside the data block are left unharmed until overwritten.

Table.rs

Struct Table:

* Represents a key-value table that is located on Pravega implemented on the user client. Contains a name meant to represent the unique table, the factory on the client side, and a token provider for the Pravega server.

Enum TableError:

* An Enum that binds different errors to different messages. Errors accounted for are connection error (between the client and server through the table), key does not exit, table does not exist, incorrect key version, and operation error.

**Error Module**

error.rs

* An Enum that is used to Conditional Check failure, Internal failures, an invalid inputs.

**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#.

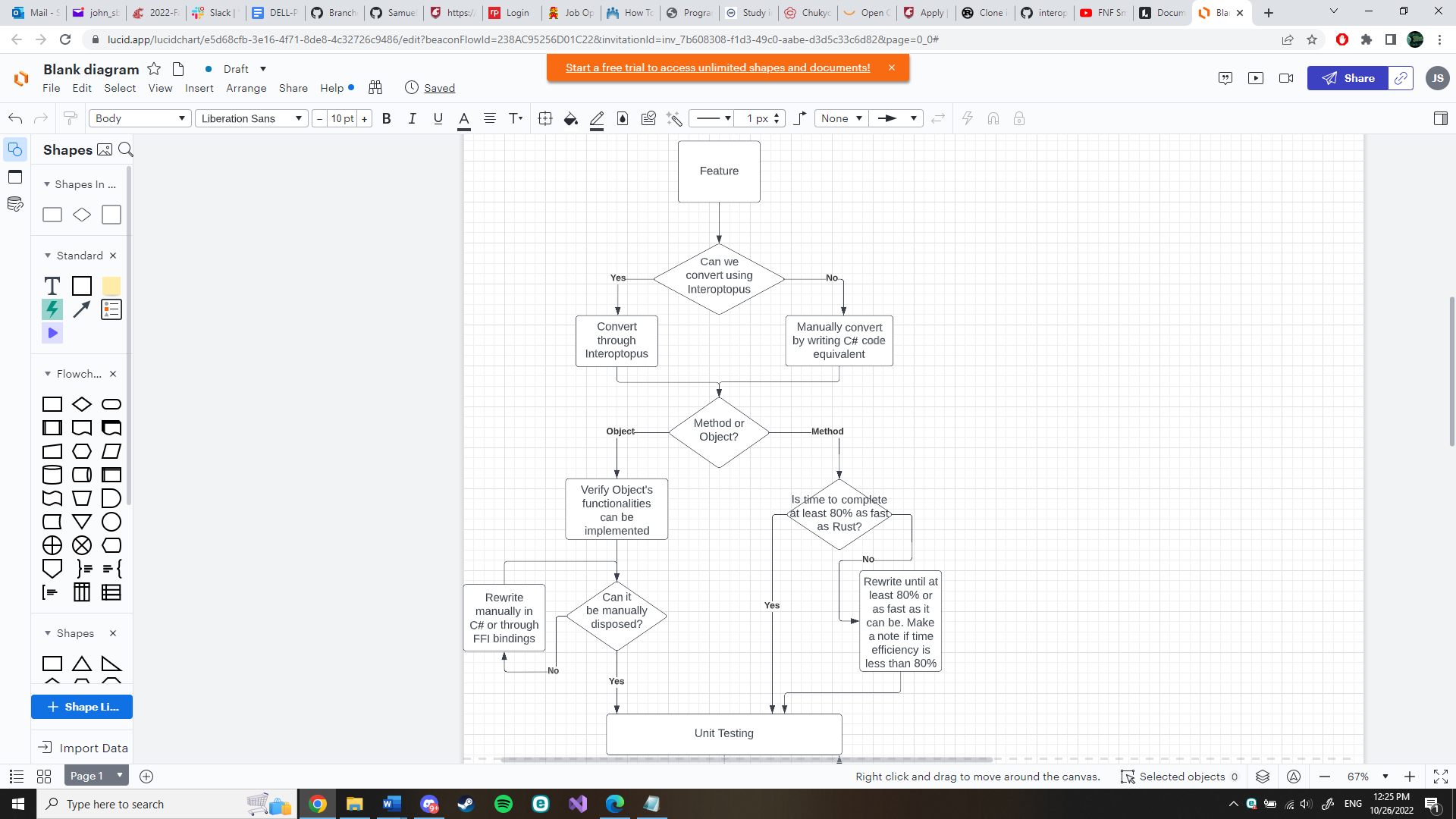
The major functionalities that we would like to test are being able to read and write events, being able to write, commit, and abort transactions, being able to read and write bytes, being able to synchronize states and tables, and being able to manage streams.

**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**

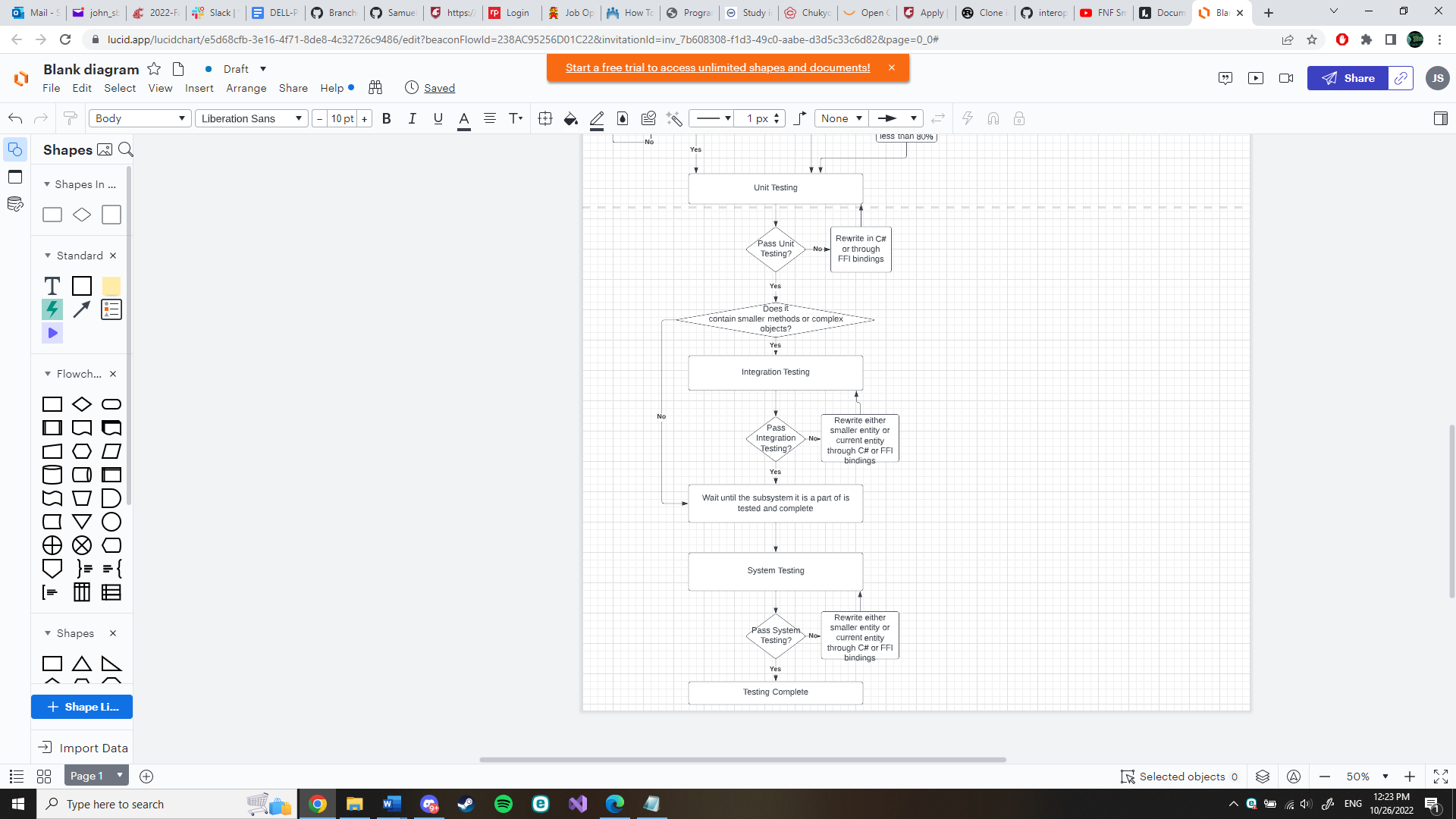
This project is about seeing how well features and functionalities in Pravega from Rust transfer from Rust to C#. As such, our testing involves seeing how well these features transfer over well. Features can be split into two categories, objects and methods.

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.

**Alpha Prototype Report**

# **Alpha Prototype Description**

We have fully completed the Rust testing parts. We spent several weeks testing 40 Rust features to see if they would transfer over to C#. Many features we we’re able to fully transfer over or partly transfer over. For most of the features we could not successfully transfer over, we found workarounds by using features that could. A good example of this is we will be using slices in place of arrays and vectors. Every feature tested has been documented in an excel spreadsheet explaining the workaround or problem if a feature does not easily cross over.

After we completed the Rust testing parts, we started work on transferring over the Pravega client Rust code. Our goal for the prototype is to have a client that can produce a Pravega instance and write to the stream. The plan for transferring over the Pravega client is to convert any functions, methods and structs into C# code that are needed to interface with the Pravega client. We will have intermediary functions to help ensure the Rust code returns and receives values that our wrapper library (Interoptopus) can handle. The first step in transferring the Pravega client is to convert the client factory. We are currently part way through converting all the struct of Client Factory over to C#

We have recently started to change our approach on how we are converting the Client Factory code and future sections. Instead of having intermediary functions we are now planning on carrying over variable data from the bytes. We are still researching effective ways to go about this and have no code written with this new approach so far.

So far, we have broken down the client factory into methods and structs that must be converted to C# and have started to convert the methods and structs over.

# **I.1 [Rust Component Testing]**

# **I.1.1 Function and Interfaces Implemented**

Implemented all Rust components test. There is no other work needed currently for the tests.

# **I.1.2 Preliminary Tests**

Since this part was building tests, all the tests included was the functionality implemented

# **I.2 [Client Factory Subsystem]**

# **I.2.1 Function and Interfaces Implemented**

Our team is currently working on transferring over the client factory subsystem. The remaining work is converting the methods and structs shown below in the UML diagram.

# **I.2.2 Preliminary Tests**

No tests so far built for this subsystem.

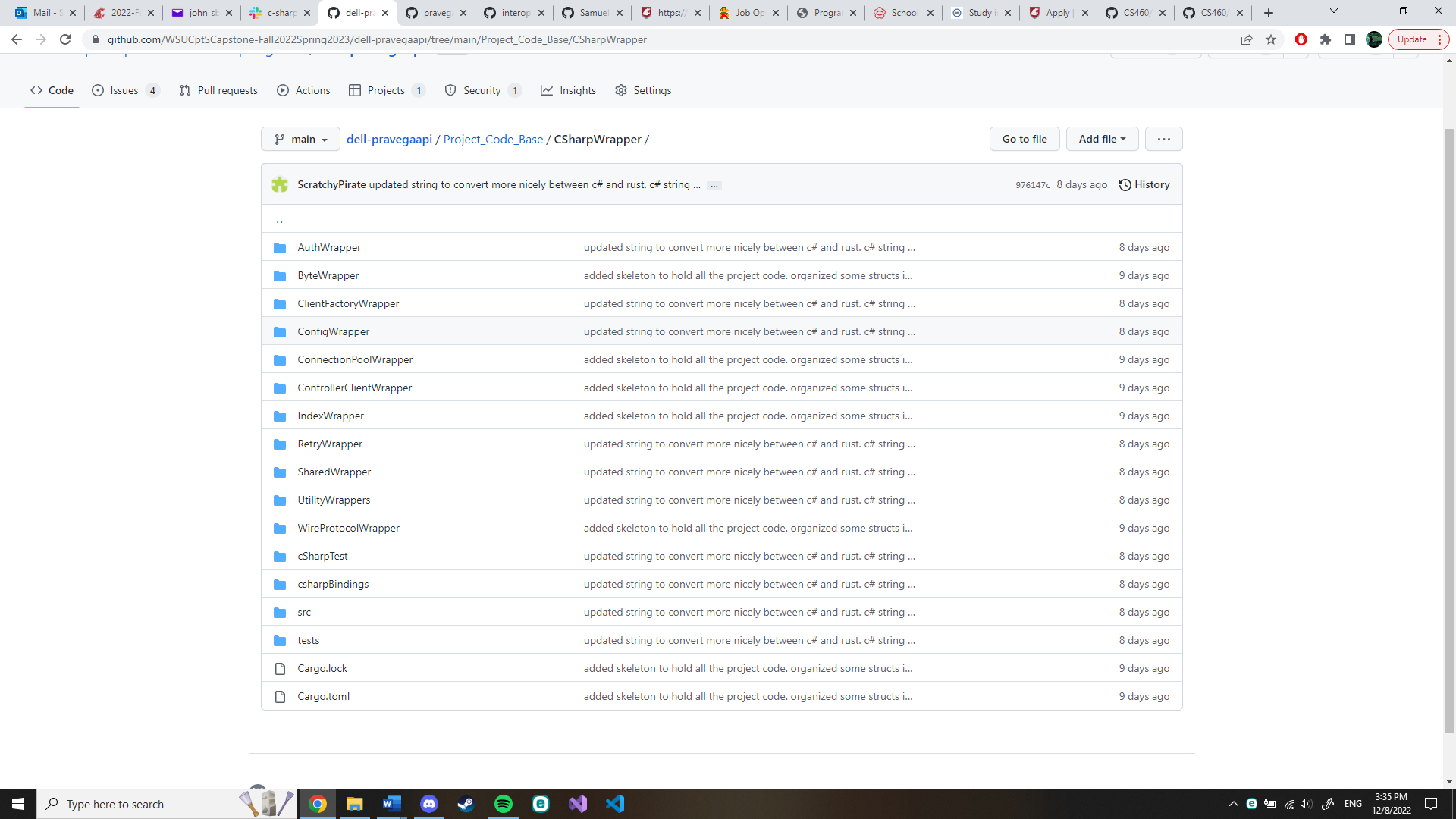
Diagram

Description automatically generated

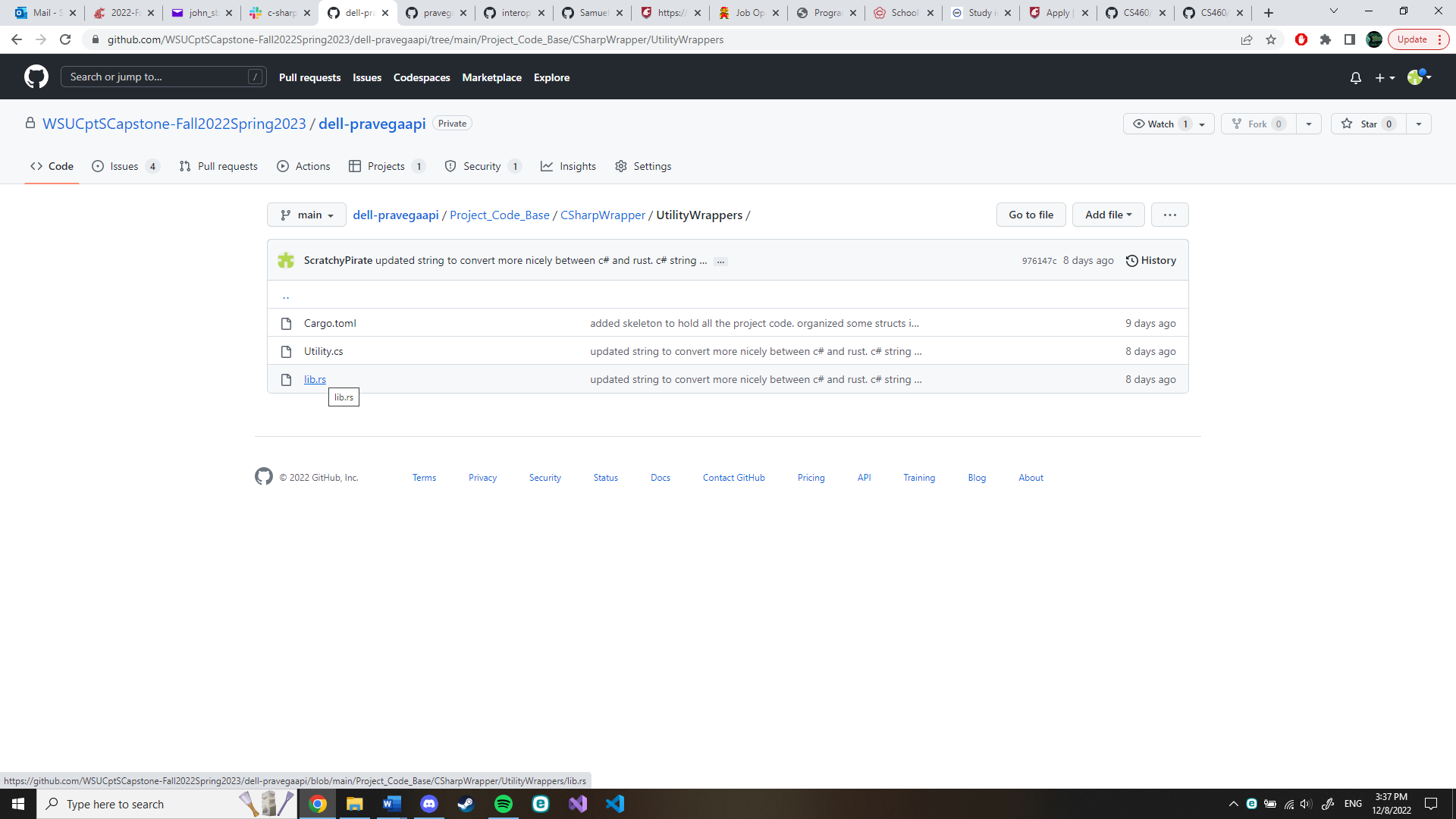
1. **Alpha Prototype Demonstration**

The main idea behind our prototype is to show a proof of concept, that Rust Pravega code can be bridged or manually implemented and still maintain functionality within the C# environment.

As of writing this, our prototype is not fully complete. We weren’t able to finish according to schedule due to design constraints we encountered while working. However, there are things that are complete for demonstration. The code skeleton is a large part of the prototype and contains setup for the C# library as well as files that are involved in wrapping the Rust side into Pravega’s Rust code.



This code skeleton is set up in a way that reflects our original solution approach architecture and reflects the architecture of Pravega’s main Rust repository, leading to easier development for us but also easier future development after this project.



Inside each folder is the C# and Rust source files that handle the communication between the two languages. The C# side also contains the library that a C# developer will be using at the end of this project and uses the namespace architecture.

As a smaller part of converting the Client Factory, the utilities area we added serves as an area for compiling smaller components represented uniquely in Rust that need to be transferred over and aren’t part of Pravega specifically. This includes things like vectors, strings, and other complex data structures with the possibility of smaller functions. This next section of code represents a full conversion of a string from Rust to C# as an example. This process also works in reverse. The core idea is to break down the complex string data structure into smaller components and transfer what we can through the languages, reassembling on the other side.

|  |  |
| --- | --- |
| public partial struct CustomCSharpString |  |
| { |  |
| public ulong capacity; |  |
| public U16Slice string\_slice; |  |
| } |  |
| public partial struct CustomCSharpString |  |
| { |  |
| // Default constructor. Creates Custom CSharp string from C# string |  |
| public CustomCSharpString(string source){ |  |
|  |  |
| // Set up a slice for the CSharp string using Marshal. |  |
| U16Slice source\_slice; |  |
| source\_slice.slice\_pointer = Marshal.StringToHGlobalUni(source); |  |
| source\_slice.length = (uint)source.Length; |  |
|  |  |
| // Set this object's string slice to the source\_slice |  |
| this.string\_slice = source\_slice; |  |
| this.capacity = source\_slice.length; |  |
| } |  |
|  |  |
| // Constructor from CustomRustString. |  |
| public CustomCSharpString(CustomRustString source){ |  |
|  |  |
| // Grab all the bytes of the source |  |
| byte[] source\_utf8\_bytes = new byte[source.capacity]; |  |
| int i = 0; |  |
| foreach (byte utf8Char in source.string\_slice){ |  |
| source\_utf8\_bytes[i] = utf8Char; |  |
| i++; |  |
| } |  |
|  |  |
| // Convert utf8 byte array into utf16 byte array |  |
| byte[] unicode\_bytes = Encoding.Convert(Encoding.UTF8, Encoding.Unicode, source\_utf8\_bytes); |  |
|  |  |
| // Convert utf16 byte array into a string |  |
| string translated\_utf16\_string = Encoding.Unicode.GetString(unicode\_bytes); |  |
|  |  |
| // Convert this string into all the pieces necessary for a //customCSharpString |  |
| CustomCSharpString new\_custom = new CustomCSharpString(translated\_utf16\_string); |  |
| this.capacity = new\_custom.capacity; |  |
| this.string\_slice = new\_custom.string\_slice; |  |
| } |  |
|  |  |
| // Convert back from slice into C# string |  |
| public string ConvertToString(){ |  |
|  |  |
| // Add each element in the slice to a string. |  |
| string returnString = string.Empty; |  |
| foreach (ushort element in this.string\_slice){ |  |
| Console.WriteLine(((char)element).ToString()); |  |
| returnString += ((char)element).ToString(); |  |
| } |  |
|  |  |
| // Return compiled string |  |
| return returnString; |  |
| } |  |
|  |  |
| // Convert from Custom C# string into Custom Rust string |  |
| public CustomRustString ConvertToRustString(){ |  |
|  |  |
| // Check to make sure this isn't empty. If it is, return empty |  |
| CustomRustString returnObject = new CustomRustString(this.capacity); |  |
|  |  |
| // Parse through and set array contents of utf8 to this string's contents |  |
| Encoding utf16 = Encoding.Unicode; |  |
| Encoding utf8 = Encoding.UTF8; |  |
| byte[] source = Encoding.Unicode.GetBytes(this.ConvertToString()); |  |
| byte[] translated = Encoding.Convert |  |
| ( |  |
| utf16, |  |
| utf8, |  |
| source |  |
| ); |  |
|  |  |
| // Assign to utf8 rust string and return afterwards |  |
| int i = 0; |  |
| foreach (byte utf8Char in translated){ |  |
| returnObject.string\_slice[i] = utf8Char; |  |
| i++; |  |
| } |  |
| return returnObject; |  |
| } |  |
| } |  |

In addition, on the Rust side is a segment of code used to pick up what this structure is outputting.

|  |
| --- |
| // Custom string used for transferring |
| #[ffi\_type] |  |
| #[repr(C)] |  |
| pub struct CustomRustString { |  |
| pub capacity: u32, |  |
| pub string\_slice: U8Slice, |  |
| } |  |
| // Functions related to CustomRustString |  |
| impl CustomRustString{ |  |
|  |  |
| // Default constructor |  |
| pub fn new(new\_length: &u32) -> CustomRustString{ |  |
| CustomRustString{ |  |
| capacity: \*new\_length, |  |
| string\_slice: U8Slice::new(new\_length) |  |
| } |  |
| } |  |
|  |  |
| // Construct a string from a normal rust string. |  |
| pub fn from\_string(source\_string: String) -> CustomRustString{ |  |
| let string\_size: usize = source\_string.len() as usize; |  |
| let mut source\_string\_clone: String = source\_string.clone(); |  |
| unsafe{ |  |
| return CustomRustString |  |
| { |  |
| capacity: source\_string.capacity() as u32, |  |
| string\_slice: U8Slice::from\_rust\_u8\_slice\_mut(source\_string\_clone.as\_mut\_vec().as\_mut\_slice(), &string\_size), |  |
| } |  |
| } |  |
| } |  |
|  |  |
| // Convert the contents of the object into a normal Rust string |  |
| pub fn as\_string(&self) -> String{ |  |
| return String::from\_utf8\_lossy(self.string\_slice.as\_rust\_u8\_slice\_mut()).to\_string(); |  |
| } |  |
| } |  |

This pattern is used for all components and functions in general up until this point for our C# wrapper between the two languages.

The functionality of Client Factory involves the creation of readers and writers and the closing of said creations [14]. While we weren’t able to implement this functionality completely, we were able to implement many of the smaller components of Client Factory fully. These include the components ClientConfig, ScopedStream, Stream, WriterID, Scope, ScopeSegment, DelegationTokenProvider, and SegmentMetadataClient. In the case that structures weren’t implemented since they would be covered in a future sprint, placeholders were put in their place. For example, ByteReader and ByteWriter were added as placeholders instead of fully implemented since they will be implemented when our team covers the Byte module.

**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)