Pravega Client Library for .NET

Project Requirements and Specifications

Sponsored by: Dell Technologies

Icon

Description automatically generated

**By: The Vegateers**

Sbur, John

Lopez, Samuel

Cook, Brandon

**I.\_Introduction**

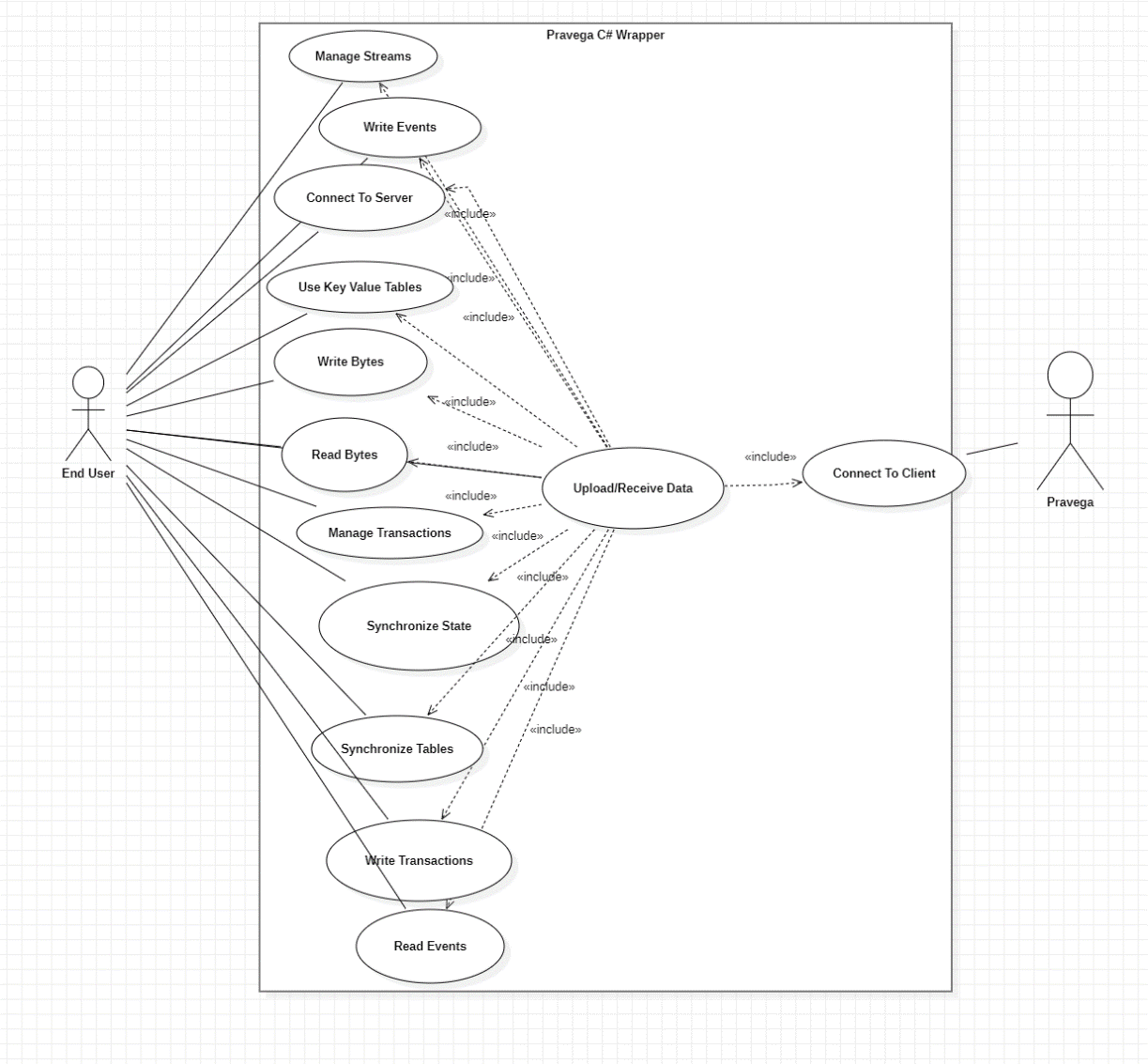
Pravega is an open-source storage system implemented and led by Dell Technologies. It uses Streams as a first-class primitive which are based on the append-only log data structure. They are flexible and have good performance [1]. By implementing clients for Pravega in multiple languages, its use can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



**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 [2]

**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 [2]

**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 [2]

**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.[4]

**Source:** Supported from past Pravega API wrapping projects [2]

**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 [2]

**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.[3]

**Source:** Supported from some of the past Pravega API wrapping projects [2]

**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.[3]

**Source:** Supported from some of the past Pravega API wrapping projects [2]

**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. [4]

**Source:** Supported from Pravega API written in Java [2]

**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. [6]

**Source:** Supported from Pravega API written in Rust [2]

**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 [2]

**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.[6]

**Source:** Supported from past Pravega API wrapping projects [2]

**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.

**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: 20-Sep-2022].
* [2] “Supported APIs · pravega/pravega-client-rust Wiki,” *GitHub*. https://github.com/pravega/pravega-client-rust/wiki/Supported-APIs (accessed Sep. 28, 2022).
* [3] “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).
* [4] “Transactions - Exploring Pravega,” *cncf.pravega.io*. https://cncf.pravega.io/docs/nightly/transactions/ (accessed Sep. 28, 2022).
* [5] “Client APIs - Exploring Pravega,” *cncf.pravega.io*. https://cncf.pravega.io/docs/nightly/javadoc/ (accessed Sep. 28, 2022).
* [6] “pravega\_client::sync - Rust,” *docs.rs*. https://docs.rs/pravega-client/0.2.0/pravega\_client/sync/index.html (accessed Sep. 28, 2022).
* [7] “Rust client for Pravega,” *GitHub*, Aug. 01, 2022. https://github.com/pravega/pravega-client-rust/blob/master/book/src/Python/PythonBindings.md (accessed Sep. 28, 2022).