Pravega Client Library for .NET

Testing and Acceptance Plans

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 [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 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 it’s team members, our solution approach, and the requirements and specifications.

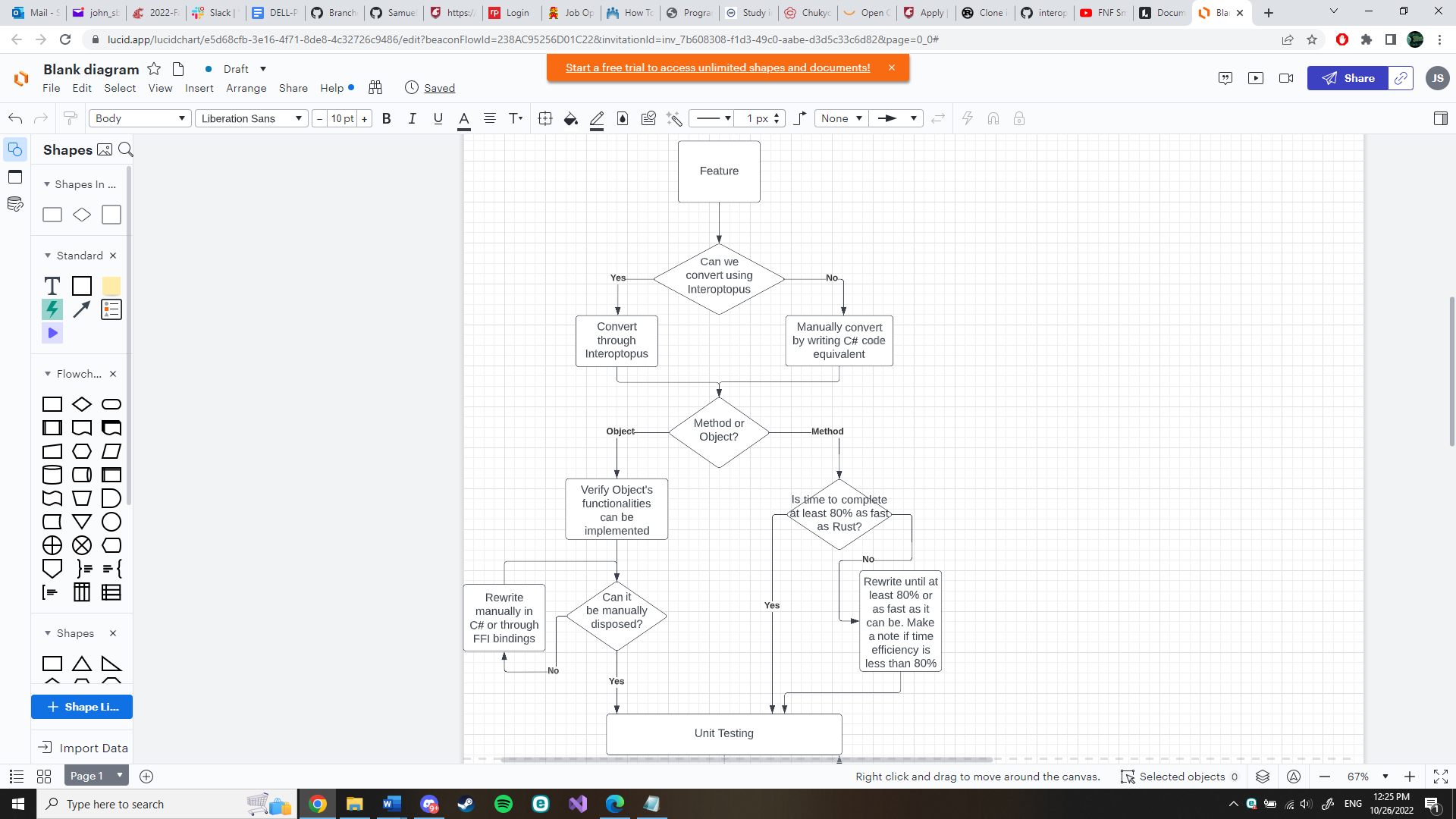
**II. Testing Strategy**

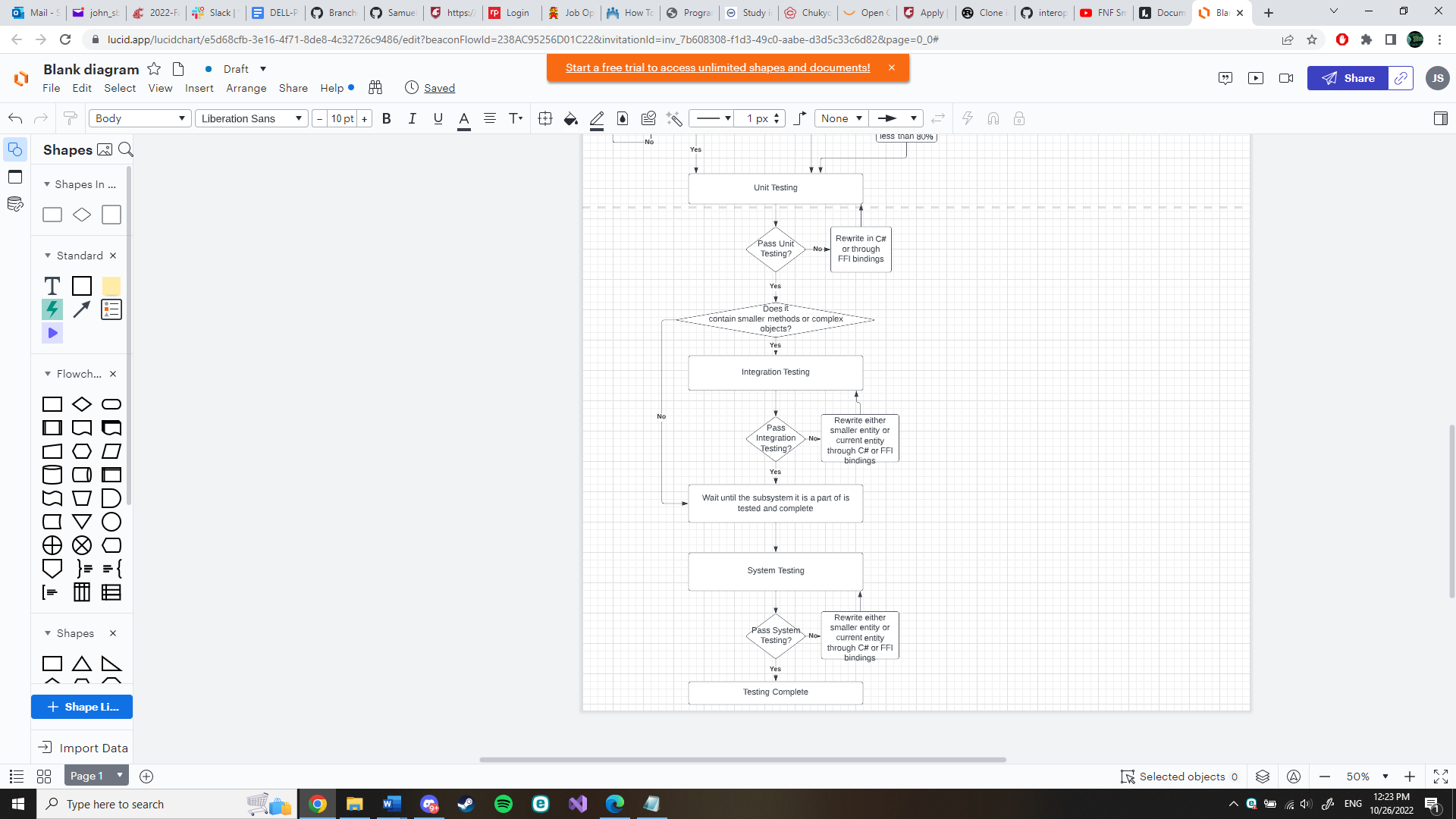
This project is about seeing how well features and functionalities in Pravega from Rust transfer from Rust to C#. As such, our testing involves seeing how well these features transfer over well. Features can be split into two categories, objects and methods.

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. 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. 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. 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.

**References**

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