Solution Approach CS421

**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.

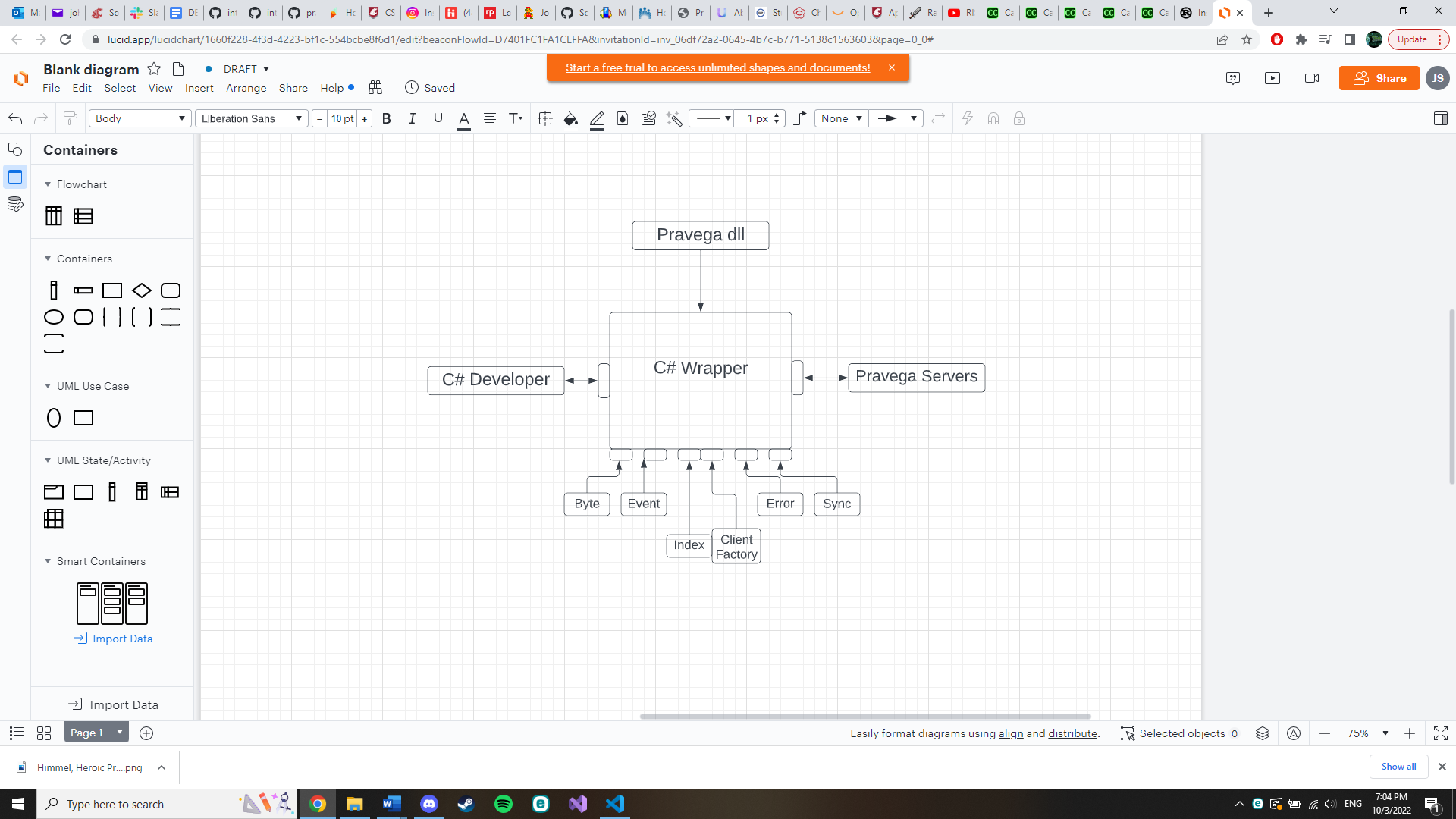
The way this project is designed to convert the modules and their content through the wrapper is by using an FFI, a mechanism that lets one programming language use code from another, to generate the code in C and then generate it into C#. The FFI we have selected for this project is Interoptopus, as 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**



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**

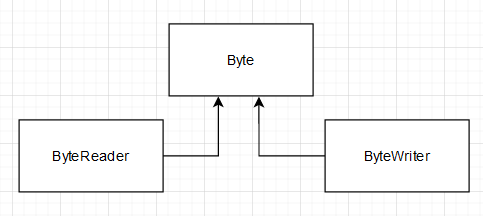
This subsystem is responsible for writing the byte stream correctly. The system will only let you write 8 MiB at a time. Is able to write asynchronously. The ByteWrtiter allows the user to write to the stream with bytes. You can create a new writer asynchronous read.

**Services Provided:**

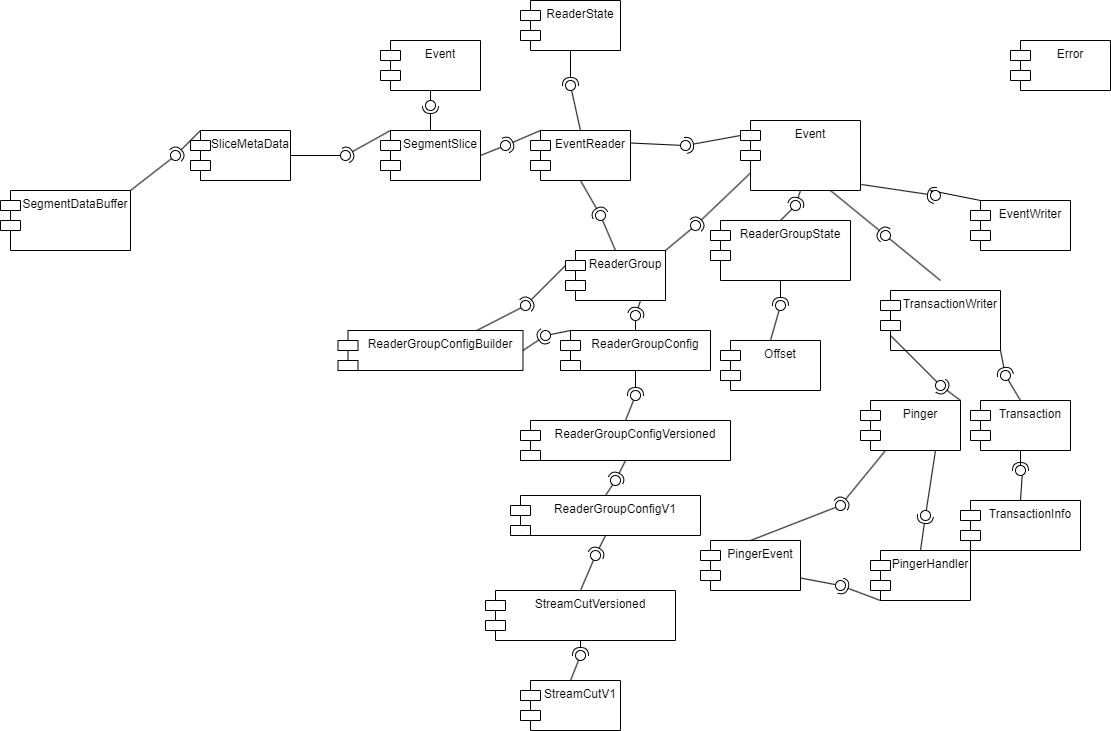
**Name**: Writer

-Provided to: C# Developer

-Description: Writes to the byte stream



**[Event]**



*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.

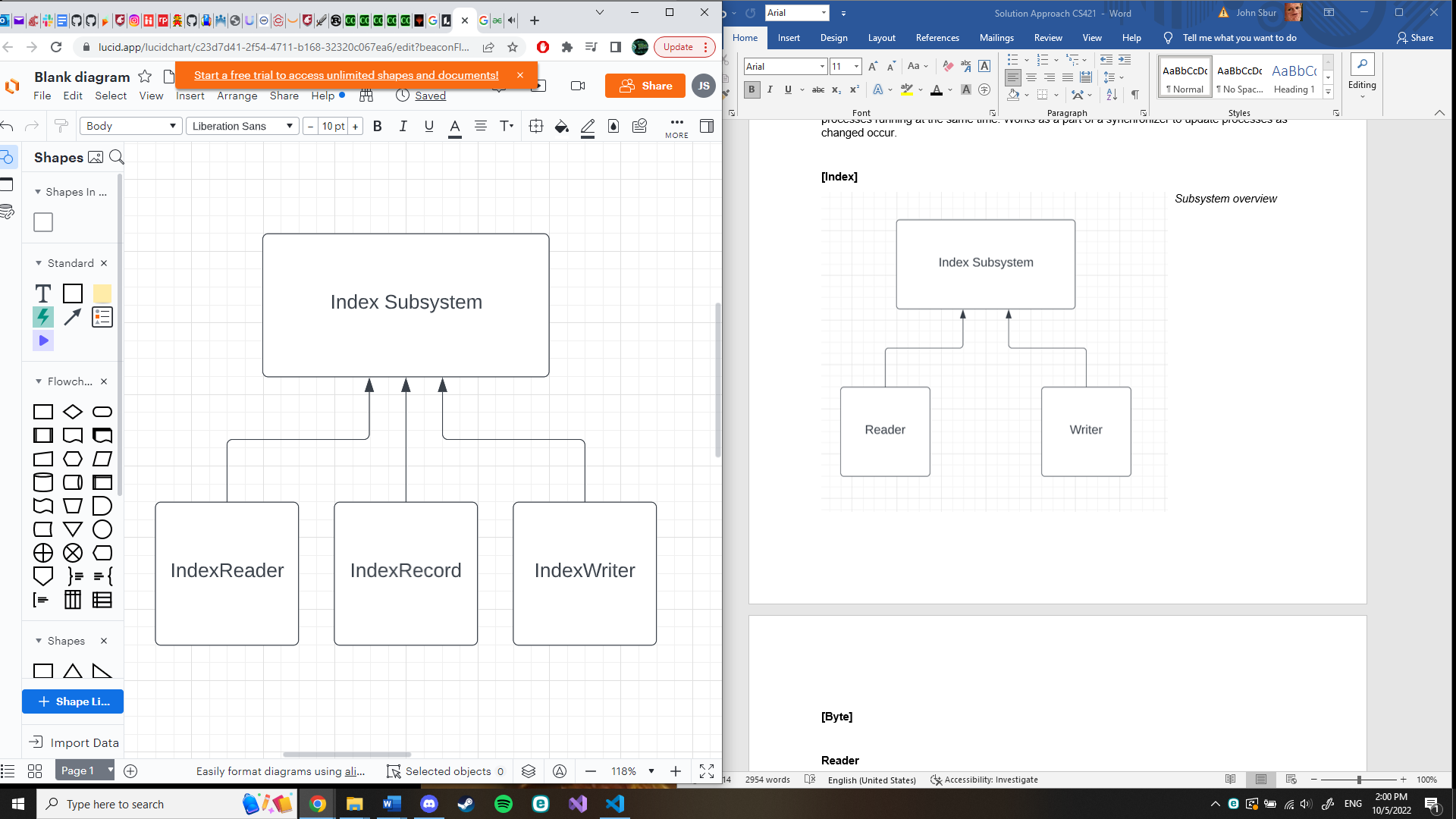
It has the functions writeEvent which writes an event and generates a routing key, writeEventByRoutingKey which writes an event with an associated routing key, flush which flushes data, and clearInitialCompleteEvent which clears completed events from the flush queue.

**Services Provided:**

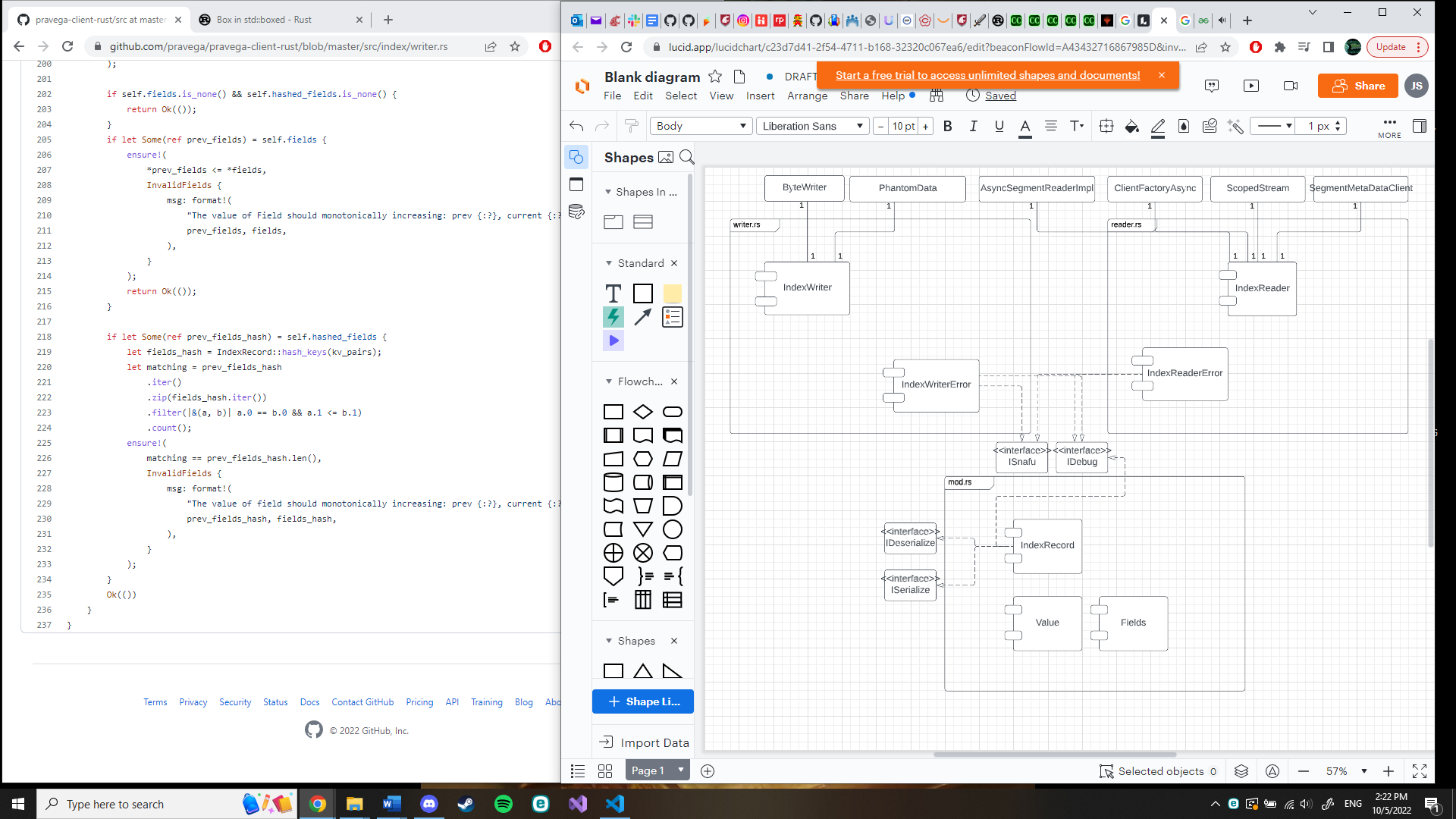
**Name:** Write Events

- Provided to: C# Developer

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

**[Index]**

*Subsystem overview*

*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**

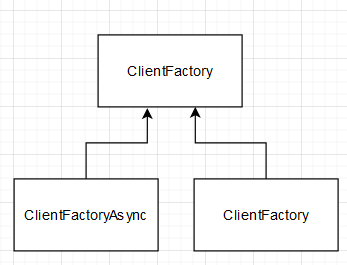
This subsection serves as an essential opposite to reader as a user is able to write user data and their fields to an IndexRecord through this class. Data can be inserted at an offset, at the beginning, or conditionally into a stream, reflecting the content in the original Pravega library.

**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]** 

**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:**

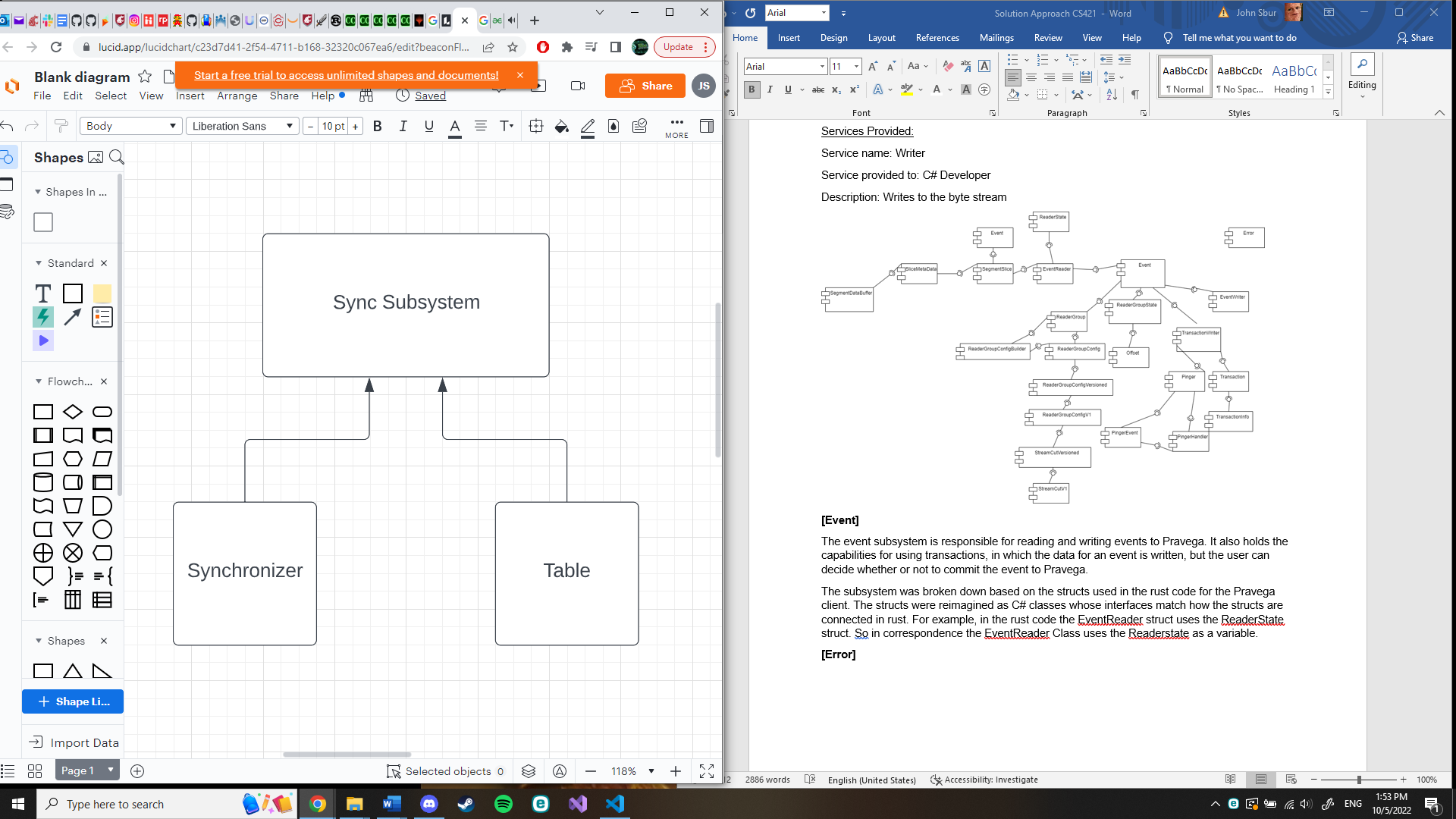
**Name**: ClientFactoryAsync

-Provided to: C# Developer

-Description: Manages client asynchronous instances

**[Sync]**

*Subsystem Overview*

**Synchronizer**

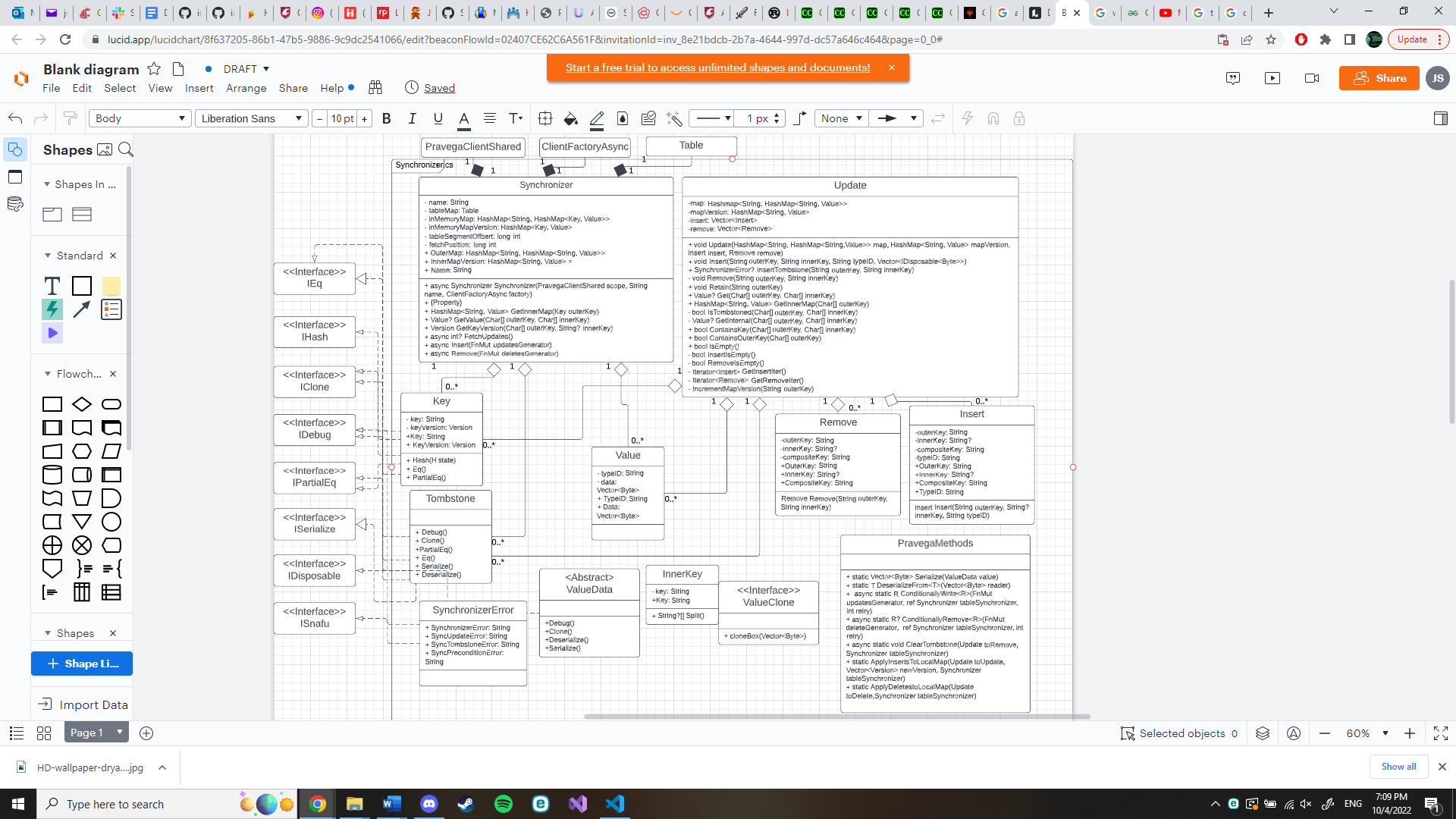
*Synchronizer.cs, a file containing the synchronizer and update control, shown above*

**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**

*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

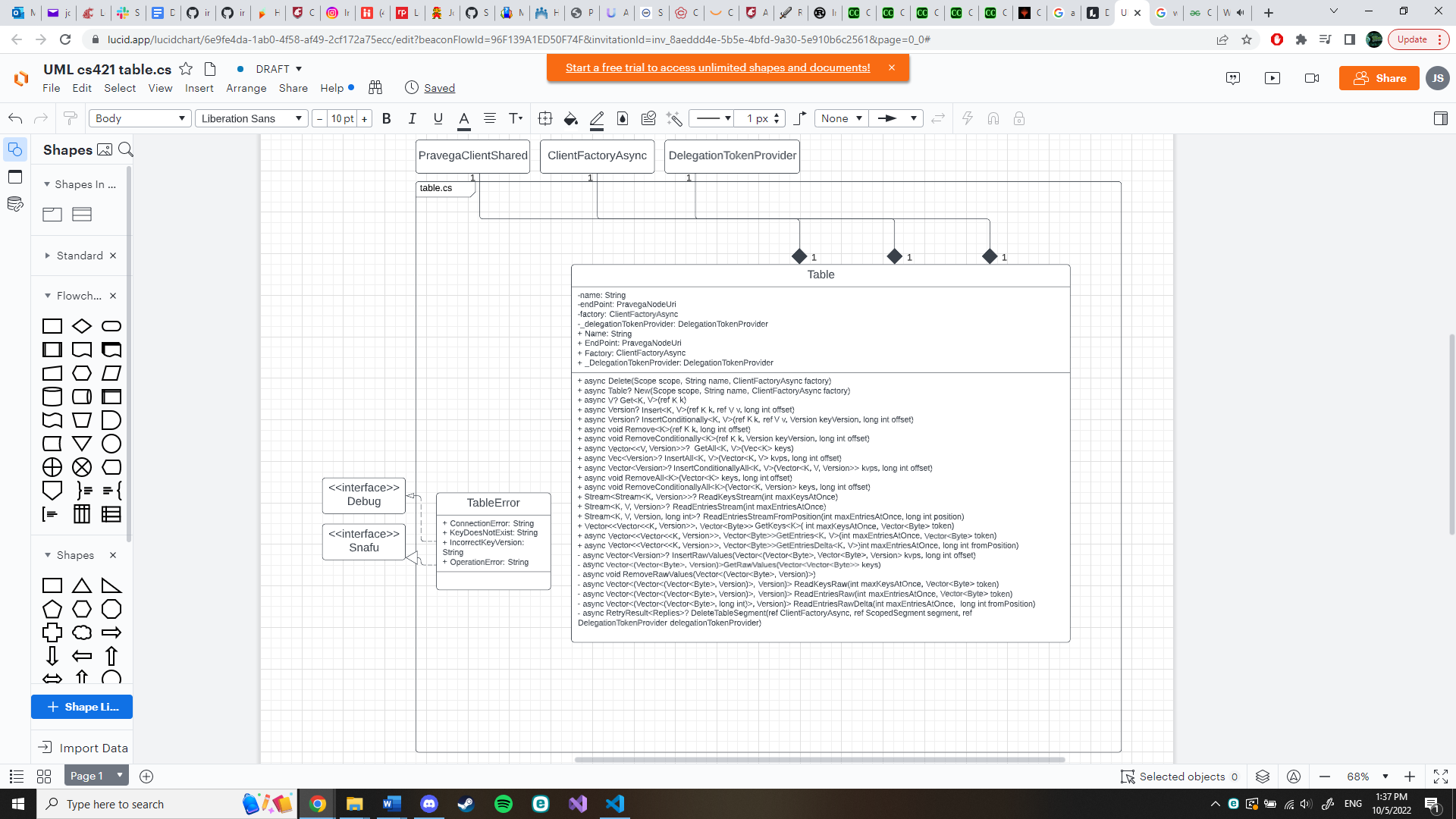
-Description: With key version control, this service lets a developer manage mapped data over multiple processes without losing accuracy. Users are able to insert and remove data from a map and have that change resonate between all other processes using that map.

**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.*

**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**

[Internal]

**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.

**References**

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