# Event Driven Object Handler Design

This presents a design that is reactive in that operations take place in response to events that they have registered with occurring. The operations are responsible for setting up their own dependencies and for obtaining required information from the overall request state as needed to determine the work they will perform. Each operation is designed to handle a small piece of work that is easy to implement and test. To create more complex operations, an operation can create multiple sub-operations and setup itself to be evented when the sub-operations complete.

They key idea is that the intelligence and how the operations interact is setup through dependencies and what the operations produce. This is a distributed intelligence model and does not have a single overall state machine that can be examined. This is slightly more complex to understand, but by distributing the intelligence, it makes it easier to validate individual or collections of operations. The design allows for easy expansion to handle different operations within the overall operation state.

The dependencies between the states are defined in code and setup at object initialization time. In other words, the dependencies are static for a particular HTTP Request type and will not change. By having the dependencies static, it allows the interactions between operations to be easily tested.

## Goals of the Design

The design intends to accomplish the following:

* Have a design that is easier to understand and maintain (a subjective goal)
* Implement a higher performance Web Server to allow higher throughput for each client connection.
* Provide a design where each piece can be individually tested and does not require the entire code base.
* Provide an initial implementation where there is a client that feeds PUT requests into a Web Server that in turn sends requests to a mock Storage Server that writes the data to a file. This provides a measurable and testable implementation that can then be expanded upon.
* Provide a design where the use of threads is explicit and easy to understand what the threads responsibility is.
* Insure that the design can be tested independently of the data transport mechanism (the design will use the NIO socket layer to communicate with clients and Storage Servers).

## Terminology

Event – For this document, an event is something that is generated when some piece of work that is interesting completes. When an event takes place, that allows a different piece of work to take place. Events and their handlers can be considered the glue that helps to define the work ordering needed to complete an HTTP request. Items that generate events include:

* Data being placed into a ByteBuffer from the SocketChannel read operation
* Data being written out of a ByteBuffer through a SocketChannel write operation
* All the data for a client object being read into buffers (meaning the entire object has been received)
* The HTTP Request from the client has been parsed and the HTTP method determined
* The HTTP Request has been sent to the Storage Server(s)
* The HTTP Response has been received from the Storage Server(s)
* All the data for a chunk has been written to the set of Storage Servers

## Object Overview

To help describe the operation of the system, there are the following objects:

* **RequestContext** – An overall object that is a placeholder for the state associated with a particular HTTP request.
* **HTTP Information** – The information pulled out of the HTTP headers and the URI.
* **BufferManager** – An object used to keep track of Buffers that has registration functions that allows “events” to be triggered when producers change the state of a buffer. The BufferManager provides the following methods:
  + register – This allows Operations to register as producers or consumers of Buffers. Consumers register as dependent upon a producer. By having the consumer registers as a dependent of a producer, that allows the Operations event handler to perform work when the producer updates a Buffer. The Operations event handler may do something as simple as mark it’s execute method as “ready” so that the EventThread will call it. Or it may actually perform some work and then decide what to do.
    - BufferManagerPointer – This object is used by producers and consumers to access Buffers in the BufferManager. The BufferManagerPointer is returned when the producer or consumer registers with the BufferManager.
  + unregister – This removes an Operation from the BufferManager.
  + offer – This is how a producer updates the BufferManager and will trigger the calls to the event handlers that are registered as dependent upon the producer.
  + poll – Provides a consumer access to a Buffer and updates it’s BufferManager pointer. It returns null if the consumer’s BufferManagerPointer is the same position as the producer’s BufferManagerPointer that it depends upon
  + peek – Provides a consumer access to a Buffer without updating its BufferManagerPointer. It returns null if the consumer’s BufferManagerPointer is the same position as the producer’s BufferManagerPointer that it depends upon.
  + bookmark – Add a placeholder in the BufferManager that can be used to distinguish operations on different chunks. This allows a new operation that is a consumer to pick a particular location within the buffer manager to begin pulling data from.
* **Buffers** – A generic term for an object that has a ByteBuffer backing it. This is used to handle the clients request information and data coming in from the wire.
* **Operations** – An Operation implements a common interface and provides the building blocks to perform requests. The methods within the Operations perform work on the Buffers or results generated from the buffers.
  + **Operation Methods** – Operations provide the following methods, though not all methods are required (meaning certain methods may not perform any work).
    - initialize – This sets up all preconditions for the Operation to run.
    - event – This is the method that executes when some other pre-condition is met that allows the Operation to run.
    - execute – This is the method that performs the actual work for an Operation. This work could be run on either the EventThread or a WorkerThread (in the event of a long running operation that is either CPU intensive or requires off-box resources). The execute method is placed in the “ready” state by the event handler.
    - complete – This is the last step when the execute method determines that there is no more work for this Operation to perform.
    - There are a number of other methods related to queuing the Operation onto the work queue for the RequestContext. These can be changed to something more efficient than actually adding them to a queue (with its associated locking requirements), but it makes the code easier to debug for the short term. The methods related to queuing are:
      * isOnWorkQueue
      * isOnTimedWaitQueue
      * markAddedToQueue
      * markRemovedFromQueue
      * hasWaitTimeElapsed – For use with the timed wait queue
  + Example Operations include:
    - Parse HTTP Request – This processes Buffers and produces HTTP headers and URI information. It completes when all the headers have been read in. It produces information from the headers and the URI that is stored in an HTTP Information object that is associated with the RequestContext. It will also determine the operation type (i.e. PUT, GET, etc.).
    - MD5 Digest – This will take in Buffers and compute the Md5 digest from them. When it processes all the buffers (for a PUT operation this is when the entire content length amount of data has been processed) it will send an event to the Compare MD5 operation.
    - Authenticate – This will use the HTTP Information to authenticate the request.
    - Encrypt – This encrypt the Buffers read in and encrypt them and place the result in a new Buffer.
    - Write To Storage Server – This will take Buffers generated from the Encrypt operation and write them out the Storage Server in chunk sizes.

## Thread Overview

There are the following threads in the initial design:

* **Accept Thread** – This is a thread that loops on a ServerSocketChannel and a Selector waiting for the socket to be acceptable. When the socket is acceptable, it will pull out the client SocketChannel and pass that to the EventThread.
* **Event Threads** – This is a collection of threads that handles the NIO processing for one or more SocketChannel(s) and the work that is required to process the data read into the buffers. There is one loop that handles the Selector with a set of SocketChannel(s) registered. There is a second loop that executes all of the “ready” Operations for a particular RequestContext. The second loop works through all of the RequestContext assigned to the EventThread. The Operations are set to “ready” to execute based upon “events” that take place as a result of data arriving or operations on the data completing. A decision needs to be made to assign RequestContext to an EventThread or allow it to float between different EventThreads. This will have implications on the locking model for various objects.
* **Worker Threads** – This is a collection of threads used to perform CPU intensive work (MD5 digest and encryption are examples of this) or work that accesses off-box resources (Authentication or VON picker are examples of this).

## High Level Operation Flow

### HTTP Parsing

When there is an accept() processed for a new connection on the ServerSocketChannel, the initial steps are as follows:

1. Call registerClientSocket() for the NioEventPollBalancer. This will pick the thread the operation will be handled on. Once it picks a thread to execute on, it then calls registerClientSocket() for the NioEventPollThread.
2. registerClientSocket() for the NioEventPollThread allocates the RequestContext and an IoInterface.
3. Allocate a RequestContext and call it’s initializeServer() method. The initializeServer() method will do the following:
   1. Starting with multiple clean BufferManager(s). There are three BufferManager(s) that the RequestContext deals with:
      1. clientReadBufferManager – Used for reading data from the server side SocketChannel that the client has written.
      2. clientWriteBufferManager – Used to write data to the client. This includes the HTTP Response.
      3. storageServerWriteBufferManager – Used to accumulate data (generally after it has been encrypted) to write to the Storage Server(s).
   2. With each BufferManager, there is a fairly standard pattern that is used. For the HTTP Parser the following is setup.
      1. Register the Buffer Allocate producer – For the clientReadBufferManager, this is what adds ByteBuffers to the BufferManager (though the ByteBuffers could all be allocated at startup time) and then is responsible for doling out buffers for the read operation.
      2. Register the Read consumer that is dependent upon the Allocate producer. This lets the NIO read code know that there are buffers available to read data into.
      3. Register the Read Complete producer – This is how the BufferManager knows when there is valid data in the Buffer.
      4. Register the ParseHttpRequest consumer that is dependent upon the Read Complete producer. When the Read Complete producer updates its pointer, that generates an event for the ParseHttpRequest that tells it that it has data and can run. The PasrseHttpRequest operation will continue to pull data from available buffers until the entire HTTP Request has been parsed.
   3. Allocate at least one buffer and add it to the BufferManager. This is the Buffer Allocate producer.
   4. When there is a Buffer added, an “event” will be sent to the Read consumer saying it has a Buffer available to perform work on. For the Read consumer the “event”, will be a method that sets the OP\_READ flag in the Selector for the SocketChannel that the request’s data will be transmitted across.
4. At this point, the RequestContext has been started and the request is waiting for data to be read into the Buffer.
5. Once data has been read into the Buffer (SSL will first read data into a temporary buffer and then unwrap it into the Buffer owned by the BufferManager), this will trigger the “data ready” event. The only listener for that event at this point is the ParseHttpRequest. This will add the ParseHttpRequest execute() method to the EventThread (this is setting the Operation to “ready”).
6. The EventThread will then process any NIO tasks (i.e. read more data into Buffers). Once the NIO tasks are completed, the EventThread will then execute the “ready” Operations.
7. In this case, there is a single “ready” Operation, ParseHttpRequest. The ParseHttpRequest will feed the Buffer(s) through the HTTP parser and add the results to the HttpInformation object. If the entire request has not been parsed, it will add one or more Buffers to the BufferManager, which will kick off the read path again. At the point there is no more data to feed through the HTTP parser and it is waiting for additional data, the ParseHttpRequest is in a “not ready” state. If the entire request has been parsed, it will then determine the request type and setup the request handler. In the design here, it will only deal with the PUT request.
8. Remove the ParseHttpRequest and Read registration from the BufferManager.

The following is how a simple dot notation to describe the HTTP parsing sequence is for a single request handler. This does not account for the fact that multiple requests will be handled at the same time and will require time to be allocated to each:

while (!httpParseDone) {

Allocate.Read.ParseHttpRequest.DetermineRequestType

}

### PUT Handler

Once the parsing is completed and the HTTP method determination is completed, the PUT method will operate something like the following:

1. Perform the initialize() method on the PUT Handler. This will perform the following:
   1. The BufferManager will remain the same as for the ParseHttpRequest handler as there may be partial or complete leftover buffers from the HTTP parsing. This just means that the HTTP Request did not use an entire buffer and there may be data for the client object in the buffer.
   2. The Authentication Operation is set to the “ready” state (that assumes that this is the first operation to run following the parsing the HTTP request, but in reality, this may not be true). Note: This is not currently implemented in the initial code.
   3. At this point, there are no consumer Operations registered with the BufferManager, so there is no work for the NIO tasks. The Allocation and ReadComplete Producer Operations are still registered with the BufferManager.
2. The EventThread executes the Authentication Operation. As an aside, this could be pushed off to a different worker thread, but for the design it is not a critical item).
3. When the Authentication Operation completes (for the purposes of this, assume it is successful), this allows the next steps in the PUT operation to run. The following things will take place:
   1. Potentially add more Buffers to the BufferManager allowing more reads to take place. It is a design decision that has been left till later to determine if the ring buffer should be fully populated at initialization time or as an on demand operation.
   2. Register the Read Operation as dependent upon the Allocation Operation.
   3. Register the MD5 and Encryption Operations as dependent upon the Read Complete. Note: Currently, only the Encrypt operation is implemented and that just copies data from one set of buffers to another set of buffers (from the Client Read Buffer Manager to the Storage Server Write Buffer Manager).
   4. Register the EncryptionComplete operation as a Producer in the “Storage Server write” BufferManager.
   5. Register the StorageServerWriter Operations as dependent upon the EncryptionComplete producer.
   6. Note – There are all sorts Operations that are not being considered in this to keep the overall description simpler and easier to understand.
4. Now the various Operations can be triggered and work to process the content for the PUT operation can begin.

The dot notation for the above Operations would look something like the following (not considering the breakup of the chunk to the various Storage Servers). This does not account for the fact that multiple PUT handlers can be running at the same time:

while (!allStorageServerWritesDone) {

Authenticate.Allocate.Read.MD5.Encrypt.StartNewRequest.

DetermineVON.InitialWriterToDatabase.Shaw-256.

WriteToStorageServer.WriteShaw-256.FinalWriteToDatabase.SendFinalStatus

}

The design is such that the WriteToStorageServer is actually multiple steps and there can be multiple of those operations running in parallel and at different speeds. The steps to write to a storage server include:

* Setting up a connection to the Storage Server (the Web Server acts as an initiator).
* Sending an initial HTTP Header to the Storage Server.
* Sending the chunks worth of client object data to the storage server.
* Sending the final Shaw-256 computed value to the Storage Server
* Receiving the HTTP Response from the Storage Server and verifying that the data was received properly.

Several other things to consider, an Operation can generate an “event” the will cause another Operation to execute. One example of this is the MD5 Digest that will trigger the MD5 Compare when all of the content data has been run through the digest algorithm.

## NIO Socket Handling

The NIO socket handling layer is designed to be independent of the handling of the HTTP Request logic. The design is such that the NIO socket layer can be replaced by a different layer that obtains data from a file or just a filled in buffer. This allows the HTTP Request logic and the various operations to be easily tested without worrying about the socket handling.

The NIO Socket layer is just one implementation of the interfaces provided by the IoInterface class. The IoInterface provides the following methods:

* startClient – This is when the IoInterface is being used as a target and the information used to communicate back to the initiator is being registered.
* registerClientErrorHandler – This is how a user of the IoInterface is informed that an error has occurred that needs to be handled. For the NioSocket implementation, this could be an error that the SocketChannel was disconnected.
* startInitiator – This is when an initiator wants to open a connection to a remote device. For the NioSocket implementation this is used to communicate with the Storage Server(s).
* registerReadBufferManager – The IoInterface is designed to use the BufferManager and a BufferManagerPointer to know where ByteBuffer(s) are that are waiting to have data placed into them.
* registerWriteBufferManager – The IoInterface uses the BufferManager and a BufferManagerPointer to know where to obtain ByteBuffer(s) that have data that is to be written out.
* unregisterReadBufferManager – This removes the read BufferManager and BufferManagerPointer references from IoInterface.
* unregisterWriteBufferManager – This removes the write BufferManager and BufferManagerPointer references from the IoInterface.
* readBufferAvailable – This tells the IoInterface that a read ByteBuffer is waiting to have data placed in it. For the NioSocket implementation, this will set the OP\_READ flag for the NIO Selector.
* performRead – This is what actually places the data into the ByteBuffer. For the NioSocket, it is where the SocketChannel.read() actually takes place when the Selector says there is data waiting. For the NioSocket implementation this is called from the NioSelectHandler class.
* writeBufferReady – This tells the IoInterface that a write ByteBuffer is waiting to have data written out of it. For the NioSocket implementation, this will set the OP\_WRITE flag for the NIO Selector.
* performWrite – This is what actually performs the write from the ByteBuffer. For the NioSocket, it is where the SocketChannel.write() actually takes place when the Selector says there are available socket write buffers to hold data. For the NioSocket implementation this is called from the NioSelectHandler class.
* sendErrorEvent – This is what calls the Operation that was registered with the registerClientErrorHandler() method when an error occurs. For the NioSocket implementation this is called from the NioSelectHandler class.
* closeConnection – This is called when the connection is to be closed out and unregistered from the Select handler. For the NioSocket implementation it will also close out the SocketChannel.
* connectComplete – This is used when setting up an initiator connection and is called when the connection has been completed to the target and is ready for use.

## Operation Details

The RequestContext contains the overall state for the HTTP Request. It owns the following BufferManager(s) to perform work:

* clientReadBufferManager – Used for data coming in from the client. This includes the HTTP Request and the client object data.
* clientWriteBufferManager – For the initial implementation that only handle the V2 PUT, this is used to write the HTTP Response back to the client.
* storageServerWriteBufferManager – This holds the client object data (in an encrypted form) that is to be written to the Storage Server(s).

The following are a list of operations that take place to complete a PUT Object request. All of the error handling is not spelled out and is mostly missing. The idea is to show the dependencies between the various operations and how they could be chained together.

**Initial Request Parsing Setup**

* Requires
  + Nothing
* Produces
  + Initializes the HTTP Parsing
* Events Generated
  + Generates an event to **Allocate Buffer** (HTTP Parsing) to setup a buffer to read in the HTTP Request

**Buffer Read Metering** (for HTTP Parsing) – This can simply allow buffers to be used as part of metering if all the buffers are allocated up front.

* Requires
  + How many buffers to allocate
* Produces
  + Invalid ByteBuffer(s) in the BufferManager. This means they do not have valid data.
  + ***bufferMeteringPointer*** registered with **clientRedBufferManager**
* Events Generated
  + When a buffer is added, generate a **Read into Buffer** (HTTP Parsing) event

**Read into Buffer** (for HTTP parsing)

* Requires
  + Empty Buffers
* Produces
  + Buffers with HTTP request data
  + ***readBufferPointer*** registered with the **clientReadBufferManager** with a dependency upon the ***bufferMeteringPointer***
* Events Generated
  + When a buffer read completes, generate a **Parse HTTP Buffer** event

**Parse Http Buffer**

* Requires
  + ByteBuffer
* Produces
  + URI
  + HTTP Headers
  + Boolean - Has the entire request been parsed
  + ***httpBufferPointer*** registered with the **clientReadBufferManager** with a dependency upon the ***readBufferPointer***.
* Events Generated
  + If all the HTTP Request has not been parsed, generate a **Buffer Read Metering** event to add another ByteBuffer to allow the Read Into Buffer operation to run again an bring in more of the HTTP Request.
  + If all the HTTP Request has been parsed, generate a **Determine Request Type** event.

**Determine Request Type**

* Requires
  + HTTP Headers
  + URI
* Produces
  + What type of request handler is required
* Event Generated
  + Generate an event to the **Setup V2** **PUT Handler** – Simplified for this document to only handling the V2 PUT request.

**Setup V2 PUT Handler**

* Requires
  + HTTP Headers
  + URI
* Produces
  + Integer – Content Length
  + Sets up for the handling of the PUT Object request
    - Currently this is only the Encrypt Buffer operation
    - Will need to be expanded to computing the Md5 digest for the client object data.

**Authenticate Request (Not implemented)**

* Requires
  + HTTP headers
  + URI
* Produces
  + Boolean - Was the request valid
* Events Generated
  + If the authentication was successful, generate an event to **Allocate Buffer** to start allocating buffer to read in the client PUT object
  + If the authentication failed, generate an event to **Send Completion Status to Client** indicating the client request failed authentication

**Buffer Read Metering** (for client PUT object content read) – This may be given the entire number of buffers to read in the client object and this deals with the allocation limits and potential out-of-buffer issues. This is the same Buffer Read Metering used for the HTTP Request handling.

* Requires
  + How many buffers to allocate
* Produces
  + Invalid Buffers in the BufferManager
  + ***bufferMeteringPointer*** registered with **clientRedBufferManager**

Events Generated

* + When a buffer is added, generate a **Read into Buffer** (client object) event

**Read into Buffer** (for client PUT object content read) – This is the same Read Into Buffer used for the HTTP Request reads

* Requires
  + Empty Buffers
* Produces
  + Buffers with client object data
  + ***readBufferPointer*** registered with the **clientReadBufferManager** with a dependency upon the ***bufferMeteringPointer***

Events Generated

* + When a buffer read completes, generate an **Md5 Digest** event
  + When a buffer read completes, generate an **Encrypt** event

**Md5Digest (Not implemented) –** This can be setup to run on a different thread during registration and then the BufferManager could be made to be thread safe and then access to the buffer can take place from wherever this is running. Also, need to make sure that one connection cannot consume or queue up lots of Md5 Digest buffers to the CPU threads and introduce latency for other connections.

* Requires
  + Content Length
  + Buffer with data
* Produces
  + Md5 Digest (may be a partial digest)
  + Boolean – Has the entire digest been produced
* Events Generated
  + If the entire digest has not been produced, generate an **Buffer Read Metering** (client object) event.
  + If the entire client object digest has been produced, generate an **Validate Md5 Digest** event

**Encrypt Buffer** – This operation handles encrypting the client object data and placing the encrypted data into the BufferManager used to write the data to the Storage Server(s).

* Requires
  + Buffer with data
* Produces
  + New buffer with encrypted data
  + ***encryptInputPointer*** registered with the **clientReadBufferManager** and a dependency upon the ***readBufferPointer***.
  + ***storageServerAddPointer*** registered with the **storageServerWriteBufferManager**
  + ***storageServerWritePointer*** registered with the **storageServerWriteBufferManager** with a dependency upon the ***storageServerAddPointer***.
* Events Generated
  + If this is the first encrypted buffer for a chunk, generate a **Setup Chunk Write** event. This event will be generated when an encrypted buffer is generated that will start a new chunk.
  + Generate an event to **Compute Shaw-256** to indicate there is a Buffer to process
  + Generate an event to **Write to Storage Server** to indicate there is a Buffer to write

**VON Picker (Not Implemented)**

* Requires
  + HTTP Headers
  + URI Information
  + Chunk Number
* Produces
  + List of Storage Servers
* Events Generated
  + Generates an event to **Write Meta-Data to Database** that the meta-data write can proceed.

**Setup Chunk Write** – This sets up the write of a chunk worth of data to a Storage Server. There can be multiple chunks being written to multiple Storage Servers at the same time. So, once the information about where the Storage Servers are located (IP address and Port number) there will be a Setup Chunk Write operation run for each Storage Server. This is passed in the location information for the Storage Server the data will be written to.

* Requires
  + **storageServerBufferManager** – This is used to handle writing the HTTP Request and the Shaw-256 information to the Storage Server. It is a BufferManager that is unique to the Storage Server chunk write instance.
  + **storageServerResponseBufferManager** – This is used to handle receiving the HTTP Response from the Storage Server. It is a BufferManager that is unique to the Storage Server chunk write instance.
* Produces
  + ***addBufferPointer*** registered with **storageServerBufferManager**
  + ***respBufferPointer***registered with **storageServerResponseBufferManager**
  + This starts the connection to the Storage Server through a call to startInitiator(). The Connect Complete operation will be evented when the connection is completed.
* Events Generated

**Connect Complete** – This is run when the connection to the remote Storage Server is completed.

* Requires
* Produces
* Events Generated
  + Will event() the **Build Header to Storage Server** operation when this operations execute() method runs.

**Handle Storage Server Error** – This is run when the connection to the remote Storage Server is broken or some sort of error occurs on the connection such that the Storage Server cannot be communicated with.

* Requires
* Produces
* Events Generated

**Write Meta-Data to Database (Not implemented)**

* Requires
  + HTTP Headers
  + URI Information
  + Chunk Number
* Events Generated
  + Generates an event to **Write to Storage Server** that the writes can proceed for a particular chunk
  + Generate an event to **Write to Storage Server**

**Compute Shaw-256 (Not implemented)**

* Requires
  + Encrypted Buffer
* Produces
  + Shaw-256 value
  + Boolean - Has entire chunks Shaw-256 been produced
* Events Generated
  + Generates an event to **Write Shaw-256 to Storage Sever** when an entire chunk’s Shaw-256 has been completed

**Build Header to Storage Server** - This operation is used to build the HTTP Request to send to the Storage Server prior to sending the actual chunk data.

* Requires
  + An empty ByteBuffer to build the HTTP Request in
* Produces
  + A built HTTP Request to a particular Storage Server
  + ***writePointer*** registered with the **storageServerBufferManager**.
* Events Generated
  + When the ***writePointer*** is updated, this will cause the Write Header to Storage Server operation to be evented.

**Write Header to Storage Server** – This operation is responsible for sending the HTTP Request that was built by the Build Header to Storage Server operation. The linked dependency between the ***writeInfoPointer*** and the ***writeDonePointer*** is so this operation can be evented when the data has been placed on the wire by the SocketChannel.write().

* Requires
  + Buffer filled in with the HTTP Request information
* Produces
  + ***writeInfoPointer*** registered with the **storageServerBufferManager** with a dependency on ***writePointer***.
  + ***writeDonePointer*** registered with the **storageServerBufferManager** with a dependency on ***writeInfoPointer***.

**Write to Storage Server** – This operation runs when the ***encryptedBufferPointer*** (actually the ***encryptInputPointer*** from the Encrypt Buffer operation) is updated, meaning encrypted data has been placed into the ByteBuffer and it can now be written out.

* Requires
  + Storage Server Information
  + That meta-data information has been written to the database
  + That the HTTP Request has been successfully sent to the Storage Server.
  + Encrypted Buffer
  + The amount of data to write for the chunk to the Storage Server
* Produces
  + Boolean – Has entire chunk been written
  + ***writeToStorageServerPtr*** registered with **storageServerWriteBufferManger** (owned by the RequestContext) with a dependency on the ***encryptedBufferPointer***.
  + ***writeDonePointer*** registered with the **storageServerWriteBufferManager** with a dependency on ***writeToStoragePointer***.
* Events Generated
  + Generates an event to **Write Shaw-256 to Storage Sever** when an entire chunks data has been written to the Storage Server – Not implemented

**Write Shaw-256 to Storage Server (Not implemented)**

* Requires
  + Storage Server Information
  + Entire chunk has been written to Storage Server (Event from **Write to Storage Server**)
  + Final Shaw-256 value (Event/product from **Compute Shaw-256**)
* Produces
  + Boolean – All chunks written to Storage Server
* Events Generated
  + Generates an event to **Update Meta-Data in Database** when all the writes to a chunk have completed

**Update Meta-Data in Database (Not implemented)** – This is when the data and shaw-256 writes have completed to a Storage Server

* Requires
  + Chunk write completed
  + Number of chunks required for the client PUT object
* Produces
* Events Generated
  + Send an event to **Send Completion Status to Client** when all chunks have been written to the Storage Servers for the client object.

**Read Response Buffer from Storage Server** – This is the operation that runs to cause the HTTP Response data to be read in from the Storage Server.

* Requires
  + Empty ByteBuffer(s) which are allocated as part of the Setup Chunk Write operation.
* Produces
  + ***readBufferPointer*** registered with **storageServerResponseBufferManager** and has a dependency on ***meterBufferPtr*** (which is the ***respBufferPointer*** created in the Setup Chunk Write operation).
  + Registers the **storageServerResponseBufferManager** and the ***readBufferPointer*** with the NioSocket connection used to communicate with the Storage Server.
* Events Generated
  + Informs the NioSocket connection that there is an available read ByteBuffer that is expecting data.

**Storage Server HTTP Response Handler** – This operation handles checking that the HTTP Response returned by the Storage Server is valid and if the response was good or contained an error.

* Requires
  + At least one ByteBuffer filled in with the HTTP Response from the Storage Server
  + The Jetty HTTP Parser to perform the actual parsing of the buffer contents.
* Produces
  + ***httpResponseBufferPointer*** registered with **storageServerResponseBufferManger** with a dependency on ***readBufferPointer***.
  + StorageServerResponseCallback that is called when the parsing of the HTTP Response completes. This is how the completion and cleanup of the Storage Server chunk write is triggered.
  + The status for the Storage Server chunk write operation.
* Events Generated
  + The StorageServerResponseCallback is triggered from the HttpResponseListener when the messageCompleted() callback is invoked by the Jetty Parser.

**Validate Md5 Digest (Not implemented)**

* Requires
  + The calculated Md5 digest
* Produces
  + An error if the Md5 digest does not match the expected digest
* Events Generated
  + Generates an event to **Send Completion Status to Client**

**Send Completion Status to Client**

* Requires
  + Final Status – Meaning all the chunks have been written and meta-data all updated or an error occurred.
  + That the final Md5 digest compare was done or an error occurred
* Produces
  + Closes out the connection after the status has been sent and cleans up
* Events Generated
  + No events, just put the RequestContext back on the free listOperation Event Details for HTTP Parse

## Event Flow for HTTP Parse



Event Flow for PUT Request



## Buffer Manager Design

The Buffer Manager provides an abstraction that allows clients to register to be told when information within the buffer changes. The idea is that there is at least one producer of data (meaning something that transfer data into a ByteBuffer or allows buffers to be used) and at least one consumer of the data. For example, a producer of data could be a SocketChannel read and the consumer of the data could be the HTTP Parser. The HTTP Parser only has work to perform when the SocketChannel read has completed and updated the byte position values for the ByteBuffer. The BufferManager is designed to allow clients (producers and consumers) to register and to setup dependencies between the different producers and consumers. When the dependency is met, then the consumer is then sent an event() to inform it that there is work that can be done. This allows the design of the operations to be event driven (a reactive design) instead of a polling design (where the operations query if data is available and then perform work).

The other advantage of having the ring buffer and its associated pointers contained within the BufferManager and BufferManagerPointer objects is the different behaviors (i.e. wrap around conditions, out of filled buffers and such) can easily be tested without requiring much additional code infrastructure. Once the dependencies and how they interact are debugged, the clients simply use the methods and the code should work. It prevents having to implement the checking logic in the different Operations.



## Questions About Different Designs to Be Considered

How easily would the design accommodate switching out the NIO code and reading the buffer contents in from a file to provide a test tool?

* Meaning how much would the PUT and HTTP parse handlers need to be modified?

How easily could the design be leveraged (meaning how much code reuse could take place versus writing all sorts of new code) to write a client and a Storage Server target that would allow us to test all sorts of weird (but easy to hit in the field) scenarios?

* A client sends part of the data and then disconnects
* A client sends data at a really slow rate
* A client sends malformed data (i.e. the Md5 digest does not match)
* A client sends multiple chinks worth of data and then disconnects
* A Storage Server responds with different errors
* A Storage Server disconnects in the middle of the stream
* One Storage Server is really slow compared to the others
* A Storage Server goes away for a while and then comes back
* The Storage Server writes the data to a file that can then be compared by the client with its expected results to insure the data is not corrupted.

Maybe the argument is that we don’t need to write code to test scenarios like that and the Java test tools can handle all of that. If that is the case, I would like to understand how that can be done.

For the initial design, pre-allocation is a reasonable path to take, but assuming at some point the design needs to accommodate client throttling, where does that fit in?

* The design implies that as long as there are buffers, reads are going to take place. But, there has to be something that decides that no more reads are required (i.e. we have read in the entire content, but there are still lots of available buffers to read data into).
* Where is that decision made (meaning you can’t blindly set the OP\_READ flag based on a buffer being available)?

Since a read can request 16k (or whatever amount is decided for a good NIO buffer size), but only read in a few bytes into the buffer, what is the intent for that?

* Will the read handler loop (i.e. set the OP\_READ flag in the Selector) and read more data into the same buffer or will it pass that partial buffer off to the next phase?

What about the NIO write handler for the case when the target can only accept a piece of the write?

* How does the write handler deal with the “try again” response case for a buffer?
* Does the state machine loop on the write handler?
* In this case, I am trying to understand how the buffer management handles the buffer when a single state does not complete its processing of that buffer (NIO read and writes are great examples where this can happen) without requiring a blocking operation (in the write case, it needs to wait until the target has freed up sufficient TCP buffers that it can accept more writes).

And then few questions if the design uses a centralized decision-making state machine (I still don’t know if that is the intent of the design).

* How do you test the states in isolation to ensure that the correct “next state” is achieved without requiring a whole bunch of setup code?
* How do you easily modify the state machine to add additional steps in the future? For example, if the decision is made to add the erasure coding to the state machine, how would the design accommodate that without an entire rewrite?

The major piece this still glosses over is the state machine and how the transitions are made, which I believe it the core of the design. The threading and what the various pieces are called is minor.

I would like to understand where the decision about what state to move to is made and what that code looks like. Here are some questions:

* Where is the decision about starting a new chunk made and the associated steps required to setup the new chunk?
  + Is this in the Orchestrator?
* How are multiple chunk writes managed?
  + Does the design account for chunk writes crossing boundaries? For example, there are two Storage Servers that have completed their writes and there is one Storage Server that is slow. Are writes to the next chunk allowed to start?
  + Where does the coordination of the multiple chunk writes (i.e. writes to different Storage Servers) take place?
  + Is it possible to test just the Chunk Write code by itself?
* What does the code that manages all the decisions look like?
  + Is the intelligence distributed amongst the various states or is it contained in one place?
  + If it is contained in one place, is it a giant switch statement or a whole bunch of “if” statements?

To me, the question about where the intelligence lies is what the real design is. How does the design handle asynchronous events and run them in a synchronous manner?

It appears that certain decisions are made to simplify the design. For example, the MD5 is computed and then the encryption is done, followed by the Shaw-256 and then the write to the Storage Servers. That is a reasonable simplification, but I would still like to see the state flow and how that is encoded.

Looking at the following paragraph:

*When PutObjectHandler finally gets notification that it has volume and VON, it can relinquish pause mode and start processing data.  The sequence is MD5 – Encryption – SHA-256.  Without a doubt, each of these operations will affect ByteBuffer’s internal pointers (most notably the position pointer) so they have to be done sequentially.  PutObjectHandler will drain the pause queue into the MD5 worker thread queue.  From there the MD5 worker can feed the same buffer into the Encryption worker.  The Encryption worker has to have an output buffer provided, that should probably be done on the EventLoop thread before feeding into the MD5, but there’s probably other ways of doing this.  Finally, the encrypted buffer is fed into SHA-256 (the original buffer can be freed back to the pool in some way).  Finally, the EventThread is notified when the buffer is ready to stream to the storage servers.*

For the sentences in red, how do you intend to handle the case where the Storage Servers are accepting data at different rates? The implication was that there was a single BufferManager which did not deal with encrypted data at all.

Does the design expect that an entire Chunk write must complete to all Storage Servers before beginning any writes to another set of Storage Servers? This would be if two Storage Servers complete all the writes and the next Chunk can be worked on. Saying it is going to use Charles’ state machine to write to the Storage Servers is insufficient (in addition, he stated in the meeting his state machine does not handle timing out Storage Servers and was not a complete design). Show how the design handles writing to the Storage Servers at the same time is it reading data into the system.

How does the design handle the slow client and slow Storage Server cases? Where is the logic for this embedded.

For the following paragraph:

## **1.1        *Errors***

*If any of the workers experience errors, they can be reported via events on the EventLoop back to the PutObjectHandler.  For example, when MD5 knows it has reached the end of Content Length and does a MD5 compare, if there is a miscompare an event message can be placed on the EventLoop.*

Who handles the event? Where does that change the state of the processing? What actions are taken in response to the error?

## Throttling

I know you don’t need to use allocation to throttle, but explain to me how it will work in the design. To me, throttling is the following:

* A limit on the read rate from the client for a PUT (for a GET it is a limit on how quickly data is pushed to the client).
* A determination of how much back-pressure is being received from the various states in the system.
  + How many outstanding Md5 calculations do we want per client – We probably don’t want one client being able to dump 25MB worth of buffers into the threads that are performing Md5 digest.
    - How do we ensure fairness between different clients for access to offboard threads and CPU Intensive threads? Is there a different queue per client or do we limit how much each client can queue up to prevent large time gaps between when a client has there digest completed due to being backed up behind 20 other clients, each with dozens of buffers.
  + How many outstanding encryption operations do we want per client – Same problem as with the Md5 digest
  + How many outstanding writes do we want (meaning how many buffers are waiting to be written) to the Storage Servers?
    - Do you want to apply back pressure to the client (i.e. not perform reads) if one or more Storage Servers are slow?

I am not trying to be difficult, but I want a real explanation. Simply the cloud drawn on the board with the caption of “stuff happens here” is not sufficient. I expect the design to change, but it is pretty hard to debate the merits of something if there is not a proposal for a way to do things.

I proposed that allocation combined with Md5 are the places throttling is driven. Allocation may not be needed, but if it is accounted for up front (even if all the buffers are pre-allocated) then it allows an easy design change to be made later if we decide that certain clients are allowed more buffers than others (or we should only read X number of MB from a client per second, then allocation makes that limitation really easy based upon the ingest rate of the buffers by the reader). Since Md5 provides a nice place to determine how many more buffers are required, that can be a location (maybe not the best depending on the algorithm used) that things like back pressure (i.e. how far behind are the Storage Server writes or how many buffers are queued up to the Md5 calculation for this client) can be used to determine if the reads should be held off for a while.

HTTP Parse probably doesn’t care about throttling as the number of buffers is small and we can’t make any real decisions about the performance of a client until we process that HTTP request.