Overview

The Object Server implementation that is in this git repository (https://github.com/notterness/ObjectServer.git) is an example used to try out a number of different technologies.
The Object Server actually consists of four different services (meaning potentially independent applications that can reside at different IP addresses and different TCP Ports). The services are:

- Object Server - This is the interface that deals with the saving and retrieval of objects. This is what handles the following methods:
  - ObjectPut - Write an object to storage
  - ObjectGet - Retrieve an object
  - ObjectDelete - Delete an object
  - ObjectList - List the current saved objects
  - CreateBucket - Buckets provide a method to group objects
- Storage Server - This is what saves the pieces of the objects (chunks) onto the disk. For this implementation, each chunk is represented by a file, but in a real Storage Server, the chunks would simply be written to a block device. The Storage Servers support the following methods:
  - ChunkPut - Writes a chunk to a file
  - ChunkGet - Reads a chunk from a file
  - DeleteChunk - Deletes the file that represents a chunk (not supported in the initial submission).
- Chunk Manager - This is what manages the chunks and the various services. The services are managed through database entries (this provides a generic lookup method to access the various services and could be replaced by a version of DHCP or something to allow for easy lookup based on a server name). The chunks are associated with a Storage Server and this service provides a centralized way for Object Servers to allocate them. This supports the following methods:
  - CreateServer - This is what adds servers that manage a service to the database.
  - AllocateChunks - This allocates a set of chunks from the various Storage Servers based upon the object's storage tier.
  - DeleteServer - Remove a service from availability (not supported in the initial submission).
  - DeleteChunk - Remove a chunk from use. This will in turn call DeleteChunk on the appropriate Storage Server to clear out the backing storage (not supported in the initial submission).
  - ListServer - List all the servers for the different services.
  - ListChunks - List the chunks on the Storage Servers and their state. This allows searching by numbers different methods to limit the amount of data returned.
- Client Test - This provides numerous different test cases to validate the behavior of the various different services. It also provides the entry point for running the various services within the IntelliJ framework.

The different services can all run within three different frameworks: IntelliJ, as Docker Images and within a Kubernetes Pod. The setup of the Kubernetes Pod is currently in a different git repository (need to provide link).

The Kubernetes Pod also brings in Socat (to allow access to the Kubernetes API) and Fluent-Bit (to push logs to an external logger, in this case Elasticsearch). Once the logs are pushed to Elasticsearch, then Kibana can be used to view and search them.

The implementation uses MySQL as a backing database to store persistent data in. There are two different databases created. There is to ObjectStorageDb that is used by the Object Server to maintain information critical to its operation. There is the ServiceServerDb (not the best name) used to manage the data for the Chunk Manager. In addition to the normal table create, update and modify operations, there are examples of stored procedures in the ServerChunkMgr class.

The KubernetesInfo class provides some examples of how to access details of the Kubernetes Pod through the Java API. This is what is used to populate the lookup information about the different services when they are running within a Kubernetes POD (essentially providing a cheap version of DNS that is run once at the startup of the Pod). Currently, there is an issue with resources from the Kubernetes API not cleaning up correctly in the initial submission.

Event Driven Object Server 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 Object 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 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. There are two pieces of HTTP information related to a request, the incoming request headers and content and the outgoing response headers and content.
- **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 its 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** – The operation is defined as an interface, which may not actually be the best choice, but it does allow for some commonality in the implementation. 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. The amount of work done within the execute method depends upon the operation. Some are quite simple and others (such as WriteChunkToClient) implement a full state machine within the execute method.
        - complete – This is the last step when the execute method determines that there is no more work for this Operation to perform. This is used to tear down dependencies on buffer managers and to call completion functions.
        - 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 Chunk picker are examples what these threads could be used for).

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:
    a. Starting with multiple clean BufferManager(s). There are three BufferManager(s) that the RequestContext deals with:
        i. clientReadBufferManager – Used for reading data from the server side SocketChannel that the client has written.
        ii. clientWriteBufferManager – Used to write data to the client. This includes the HTTP Response.
        iii. storageServerWriteBufferManager – Used to accumulate data (generally after it has been encrypted) to write to the Storage Server(s).
    b. With each BufferManager, there is a fairly standard pattern that is used. For the HTTP Parser the following is setup.
        i. Register the Buffer Allocate producer – For the clientReadBufferManager, this is what adds ByteBuffer(s) to the BufferManager (though the ByteBuffer(s) could all be allocated at startup time) and then is responsible for doling out buffers for the read operation.
        ii. 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.
        iii. Register the Read Complete producer – This is how the BufferManager knows when there is valid data in the Buffer.
        iv. Register the Parse Http Request consumer that is dependent upon the Read Complete producer. When the Read Complete producer updates its pointer, that generates an event for the Parse Http Request that tells it that it has data and can run. The Parse Http Request operation will continue to pull data from available buffers until the entire HTTP Request has been parsed.
    c. Allocate at least one buffer and add it to the BufferManager. This is the Buffer Allocate producer.
    d. 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 Parse Http Request. This will add the Parse Http Request 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, Parse Http Request. The Parse Http Request will feed the Buffer(s) through the HTTP parser and add the results to the CasperHttpInfo 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 Parse Http Request 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 Parse Http Request 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 V2 PUT Handler. This will perform the following:
    a. The BufferManager will remain the same as for the Parse Http Buffer 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.
    b. 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.
    c. At this point, there are no consumer Operations registered with the BufferManager, so there is no work for the NIO tasks. The Allocation and Read Complete 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 PutObject operation to run. The following things will take place:
    a. 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.
    b. Register the Read Operation as dependent upon the Allocation Operation.
    c. 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).
    d. Register the Encrypt Buffer operation as a Producer in the "Storage Server write" BufferManager.
    e. Register the Write to Storage Server Operations as dependent upon the Encrypt Buffer producer.
    f. 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 PutObject 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 method handlers (GET, POST, PUT, DELETE, etc.) can be running at the same time:

```
while (!allStorageServerWritesDone) {
   Authenticate.Allocate.Read.MD5.Encrypt.StartNewRequest.
        DetermineChunk.InitialWriteToDatabase.ChunkMd5Compute.
        WriteToStorageServer.VerifyChunkMd5.FinalWriteToDatabase.SendFinalStatus
}
```

The design is such that the write of the chunk to the Storage Server 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:

- Determine the chunk information for which storage servers to write the chunk worth of data. This is a request to the Chunk Manager Service that manages all chunk allocations.
- Writing the chunk begin information to the *objectStorageDb.storageChunk* database table.
- Setting up a connection to the Storage Server (the Object Server acts as an initiator). There will be more than one Storage Server that will receive the chunk data.

- Sending an initial HTTP Header to the Storage Server.
- Sending the chunks worth of client object data to the storage server.
- Receiving the HTTP Response from the Storage Server and verifying that the data was received properly.
- Writing the chunk completed information to the *objectStorageDb.storageChunk* database, table to indicate the data is safely written.

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 PutObject, 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 PutObject 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 **clientReadBufferManager**
- 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.

SetupObjectPut Handler – This is what sets up the encryption of the client object data and the Md5 digest operations. It completes and events the Send Final Status operation when all of the client object data has been written to the Storage Server(s) and the Md5 Digest has completed.

- 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.
- Event Generated
  - When the Encrypt Buffer and Compute Md5 Digest operations have completed, this will event the Send Final Status operation and will clean itself up.

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 **clientReadBufferManager**

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 – 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 (will be a partial digest until all of the bytes in the client object have been processed)
    - boolean – Has the entire digest been produced
    - boolean – Was the computed Md5 Digest identical to the one passed in through the Content-MD5 header.
    - ***md5DigestPointer*** registered with the **clientReadBufferManage**r with a dependency upon the ***readBufferPointer***
- Events Generated
    - If the entire client object digest has been produced, generate a **Setup V2 Put Handler** event() to indicate it has completed.

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

Object Chunk Allocator

- Requires
    - HTTP Headers
    - URI Information
    - Chunk Number
- Produces
    - List of Storage Servers and the chunks where object data will be saved.
- Events Generated
    - Generates an event to **Write Chunk Information 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 Md5 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 Chunk Information to Database

- 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 Md5

- Requires
    - Encrypted Buffer
- Produces
    - Md5 value
    - Boolean - Has entire chunks Md5 been produced
- Events Generated
    - None

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 ***writeToStorageServerPtr***.
- Events Generated
  -

Compare Md5 to Storage Server Md5

- Requires
  - Storage Server Information
  - Entire chunk has been written to Storage Server (Event from **Write to Storage Server**) and response from the Storage Server has been received. The response from the Storage Server contains its computed Md5 value for the chunk.
  - Final Md5 value (Event/product from **Compute Md5**)
- 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 – This is when the data and Md5 comparisons 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.

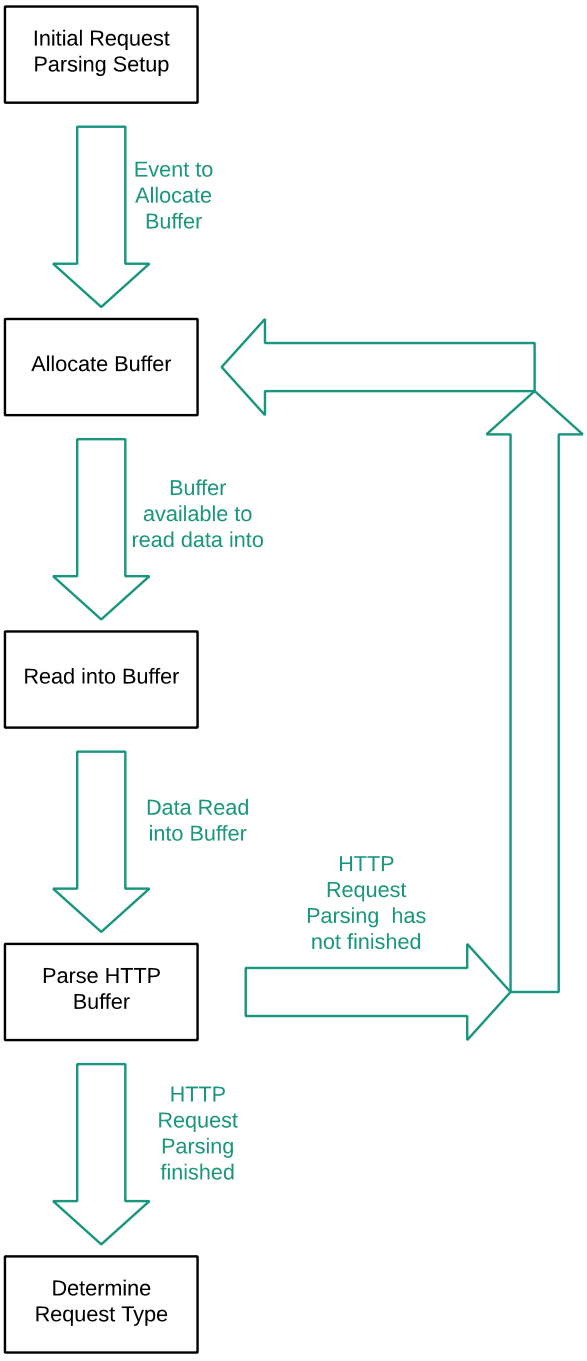
Validate Md5 Digest

- 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

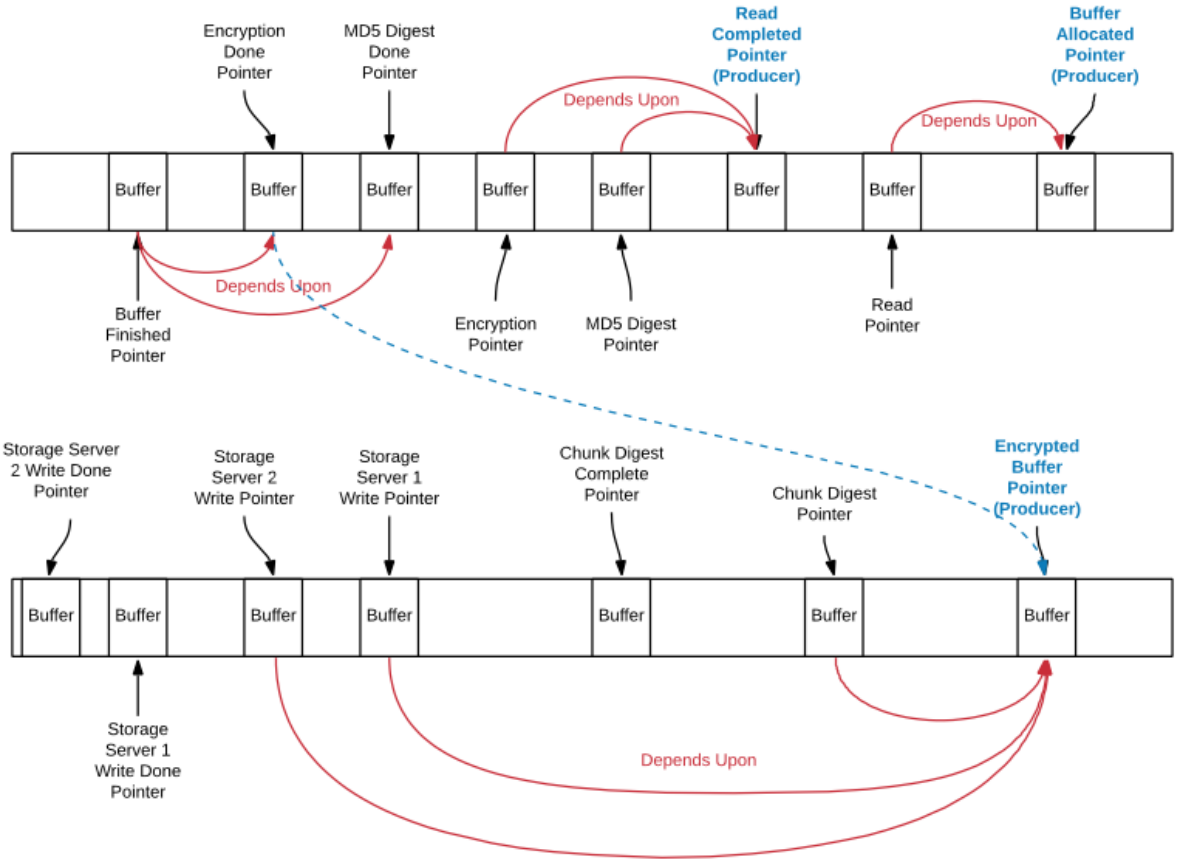


Event Flow for PutObject Request

Buffer Manager Design

The Buffer Manager provides an abstraction that allows operations 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 where operations are told to run when data is available) 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.



The two linear sequences of operations above represent a view into a two different circular buffer. The top circular buffer represents data coming off the wire from the client performing the PUT operation. The lower circular buffer represents data that was encrypted (encryption can be an an encrypt and put the result in a different buffer) and then needs to be operated on in a chunk to be sent to the Storage Servers. The "Depends Upon" represents a dependency upon a prior operation on the buffer completing prior to the next operation taking place. For example, when a "Read Completed" event takes place, that can be used to send events to the "MD5 Digest" and "Encryption" methods indicating they are ready to run. There are several features of the Ring Buffer to consider:

- Anything that depends on something else, cannot "pass" the operation it depends on in the Ring Buffer.
- If items do not have a dependency upon each other, they are free to operate at different rates. For example, "Storage Server 1 Write" and "Storage Server 2 Write" can both write to their targets at different rates.