DDN STORAGE

WOS Core
API Reference Guide

Version 2.7 | 96-30055-001 | Revision B3

C++ | Java | Python | (HTTP) RESTful

December 2016

About this Guide

This document describes how to interface an application to the WOS client API. Reasonable knowledge of the language concept and syntax is assumed.

The following APIs are available:

- C++
- Java
- Python
- (HTTP) RESTful

Background, Terminology, Concepts

The DDN WOS product permits a user to store *objects* to a *cluster* which consist of *nodes* located in various *zones*, as controlled by *policies*.

Objects contain user-specified data, and user-specified metadata. Objects are write-once, read-many. That is, once an object is written to the cluster, it cannot be modified (but it can be deleted).

The WOS **cluster** refers to the entire collection of nodes which are acting in concert to provide replicated storage of objects. Each **node** contains a CPU and some disks, and runs the WOS server software. Nodes are administratively assigned to **zones**, which typically represent distinct locations.

Clients that wish to connect to the cluster do so by referring to any of the nodes in the cluster. A web-based administrative GUI interface is also provided by the cluster.

When storing objects, **policies** specify the number of replicas to create, and in which zones to create these replicas. Policies must specify at least two replicas in one or more zones.

The Web GUI is used to create policies, and to assign nodes to zones.

When an object is written to the WOS cluster, an **OID** (object-ID) is returned to the user. This OID is representable as a printable ASCII string. The OID can be used later to retrieve a copy of the object from the cluster. OIDs are never re-used: if an object is stored with a particular OID, and that object is later deleted, the original OID no longer can be used to access the deleted object. Further, when a new object is stored in the space taken by the original object, a new OID will be returned. OIDs are a unique representation of a particular object.

A typical application will maintain a database of {user-keys} -> OID mappings.

Table of Contents

Chapter 1

C++ API

1.1 Introduction

The C++ API provides the following functionality:

- Connect to cluster
- Create WOS Objects; assign/examine object data and metadata
- `PUT`, `GET`, `DELETE`, `EXISTS` object from cluster
- `Reserve`, `PutOID`
- Objects up to 5 TB are supported
- Object may be read in pieces, using a “span” concept. Likewise, objects may be written using spans, as long as spans are contiguous, and written in-order.
- Streaming--the design of the API is such that (especially for large objects), the entire object need not be in (client) memory at one time when reading or writing the object to or from the WOS cluster.
- When using the streaming interface, object metadata can be retrieved independently of data.

Each of the `Put`, `Get`, `Delete`, `Exists`, `Reserve`, and `PutOID` calls is available in both blocking and non-blocking forms. They operate with objects with a size of < 5 TB, subject to memory availability on the client machine. (Note that this memory limit does not apply to the streaming APIs.) For example, if your computer has 2GB of memory, you cannot read or write a 3GB file with these API's. All of the public API methods are thread-safe.

Each of the calls returns a `WosStatus` status value:

- `OK` - the operation was successful
- `NoSpace` - no space available
- `ObjectNotFound` - an object specified by OID (for Get, Delete, Exists PutOID) could not be found
- `NoNodeForPolicy` - the client cannot find an active node for the specified (Put) policy
- `NoNodeForObject` - the client cannot find an active node which serves the specified (Get) object
- `UnknownPolicy` - the client specified a WosPolicy which is not currently defined
- `InternalError` - generic error

NOTE : GOA is only accessible via the RESTful HTTP API.

1.2 Connecting to the Cluster

Example:

```
#include "wos_cluster.hpp"

using namespace wosapi;

int main(int argc, char** argv)
{
    char* host = "10.0.0.2";

    try {
        WosClusterPtr wos = WosCluster::Connect(host);
        do_work(wos);
    }
    catch (WosException& e) {
        printf("WOS Exception: %s\n", e.what());
    }

    return 0;
}
```

Notes:

- Only one connection to the cluster per process may be open at one time.
- Connect is a somewhat expensive operation; it is intended that applications do this once (or infrequently) and then keep their connection open while performing `Put/Get/Delete/Exists` operations on the cluster.

1.2.1 A Note on Boost::shared_ptr

The WOS API makes use of the Boost shared library to facilitate correct code. Boost is a collection of peer-reviewed portable C++ source include files and libraries. The use of Boost in WOS is visible in the API through the use of `boost::shared_ptr<T>`'s. Each WOS type that ends in `Ptr` is `shared_ptr` to the underlying type. As such, when these objects are no longer referenced, their resources are automatically cleaned up. In the case of `WosCluster::Connect`, when the returned `shared_ptr` goes out of scope (unless it is copied), the connection to the cluster will be closed.

As a practical matter, this means that a version of Boost that defines `shared_ptr` must be installed and available to your compiler.

NOTE : WOS is qualified with Boost v1.45. WOS uses long-standing boost shared pointer functionality, and it is expected that newer versions will not have issues, but testing with target application should be done to confirm.

Boost libraries can be found at www.boost.org.

1.3 Creating WosObjects

Prior to storing data to the WOS cluster, data and its associated metadata must be bound with a `C++ WosObject`. Data is described with a `(pointer-to-start, len)` pair, while metadata is described by `(key, value)` pairs.

Example:

```
using namespace wosapi;
...
WosObjPtr create_object(void* data, size_t len, const char* type)
{
   WosObjPtr obj = WosObj::Create();

   obj->SetData(data, len);
   obj->SetMeta("Content-Type", type);

   return obj;
}
```

1.4 Retrieving Data and Metadata from WosObjects

Data is retrieved from an object via a (`pointer-to start, len`) pair. The memory block referred to by this pair is valid until the associated WosObjPtr is released.

Metadata can be accessed via known keys. Additionally, a *visitor-style callback api* is provided for examining all metadata.

Example:

```
using namespace wosapi;
using namespace std;

// forward declaration:
void meta_visitor(void* context, const string& key, const string& value);

void get_data(WosObjPtr obj)
{
    const void* base;
    size_t len;

    if (obj) {
       obj->GetData(base, len);           // returns base, len

       string color;
       obj->GetMeta("color", color);      // returns color

       process_data(base, len, color);    // e.g. (user-supplied)
     }

    // Use "EachMeta" to print values of each metadata entry...
    if (obj)
       obj->EachMeta(0, meta_visitor);       // context is not used...
}

void meta_visitor(void* context, const string& key, const string& value)
{
    printf("Metadata key: %s, value: %s\n", key.c_str(), value.c_str());
}
```

Notes:

- The maximum size of user-defined metadata is 64MB.
- Objects (data and metadata) are write-once. Once an object has been stored, and an OID is returned for that object, the object cannot be modified. [1]

1. A copy of the object (or a modified object) can also be stored, resulting in a new OID; after suitable transformation of the applications lookup database, the original OID/object can be deleted.

1.5 Put / Get / Delete / Exists Overview

The basic operations on a WOS cluster are simple:

- Objects can be stored using the `Put` call; this call returns an `OID`, which is later used when retrieving the object.
- Objects can be retrieved by `OID` using the `Get` call.
- Objects can be permanently removed from the cluster with the `Delete` call.
- Objects can be checked for existence using its OID.

Additionally, it is possible to reserve an `OID` for an object to be stored later using the `Reserve` call. This call returns an `OID` that does not yet refer to an object which can be used exactly once to store an object using the `PutOID` call. This scenario may be useful in applications which want to update their database (`key`, `OID`) mapping prior to delegating the storage of an object to a third party.

> **NOTE :** Reservations consume a small amount of disk space and count toward the maximum object count per node. To most efficiently use your WOS resources, use or delete all reservations made.

Each of the `Put`, `Get`, `Delete`, `Exists`, `Reserve`, `PutOID` calls is available in a blocking and non-blocking form. The blocking calls wait for the operations to be complete before continuing. The non-blocking calls accept a callback function as a parameter and a user-defined context object. When the operation is complete, the callback function will be called with the results of the operations as well as the user-defined context.

To maximize WOS performance:

- For single-threaded applications, use the non-blocking (callback-style) operations.
- For multi-threaded applications, use the blocking operations (potentially running many in parallel).

Notes:

- No more than 500 user-created threads may be created in a WOS client application
- All non-blocking callback functions are called on the thread on which you connected to the cluster.
- The “wait” command (used to block until all outstanding non-blocking requests have been completed) must be called on the thread on which you connected to the cluster.

1.5.1 Put

Blocking Put example:

```
#define P_IMAGES "images"

bool store_image(WosClusterPtr wos, void* pdata, size_t len)
{
   WosPolicy policy = wos->GetPolicy(P_IMAGES);

   // create an object; associate data, metadata with it
   WosObjPtr obj = WosObj::Create();

   obj->SetData(pdata, len);
   obj->SetMeta("key1", "value");
   obj->SetMeta("image-type", "jpeg");

   // store the object; blocking call
   WosStatus rstatus;// return status
   WosOID roid;// return oid

   wos->Put(rstatus, roid, policy, obj);

   if (rstatus != ok) {
      printf("Error during Put: %s\n", rstatus.ErrMsg().c_str());
      return false;
   }

   return true;
}
```

The non-blocking version of this code introduces a user-defined PutContext struct. Since the start of the Put is in a different function/method from the completion (callback), the context object provides a means to communicate, in application terms, between these two points.

Non-blocking Put example:

```
#define P_IMAGES "images"

struct PutContext {
   long id;
   void* pdata;
   size_t len;
};

// forward decl
void put_cb(WosStatus status, WosObjPtr obj, void* pctx);

void store_image_nb(WosClusterPtr wos, void* pdata, size_t len)
{
   WosPolicy policy = wos->GetPolicy(P_IMAGES);

   // create an object; associate data, metadata with it
   WosObjPtr obj = WosObj::Create();

   obj->SetData(pdata, len);
```

```
    obj->SetMeta("key1", "value");
    obj->SetMeta("image-type", "jpeg");

    // create a context object, which will be passed back to
    // the completion callback function.
    PutContext* ctx = new PutContext();
    ctx->id = 555;
    ctx->pdata = pdata;
    ctx->len = len;

    // store the object; put_cb will be called on completion.
    wos->Put(obj, policy, put_cb, ctx);
}

void put_cb(WosStatus status, WosObjPtr obj, void* pctx)
{
    PutContext* ctx = static_cast<PutContext*>(pctx);

    if (status != ok) {
        printf("Error during Put: %s\n", status.ErrMsg().c_str());
        delete ctx;
        return;
    }

    // At this point, the object has been successfully
    // stored to all replicas specified by the policy P_IMAGES

    // Retrieve OID associated with object:
    WosOID oid = obj->GetOID();

    // Use information in 'ctx' to update a local database:
    //update_database(ctx->id, oid);     // e.g. (user-supplied code)

    // At this point, it is ok to delete the storage associated
    // with the original (pdata, len) description of the image
    //delete (char*) ctx->pdata;          // e.g. (user-supplied code)

    // delete non-blocking context:
    delete ctx;
}
```

Notes:

- Since WOS attempts to avoid copying data when possible, the memory referenced by (`data`, `len`) *must* remain valid until the `Put` operation completes. In the case of non-blocking `Put`, the memory must remain valid until the corresponding callback function has been called.
- Once a `WosObject` has been submitted for `Put` to the cluster, it becomes *frozen*; after which, no further changes to its data or metadata are permitted.
- The `GetOID` call will only return a valid `OID` if the `Put` succeeds, that is, if `status == ok`. Otherwise, it will return a null `OID` (which will not succeed at retrieving any object, if presented to `Get`).

1.5.2 Get

An object can be retrieved from the cluster using its `OID`.

Blocking example:

```
void get_image(WosClusterPtr wos, WosOID oid)
{
   WosStatus status;
   WosObjPtr obj;

   wos->Get(status, oid, obj);        // returns status, obj
   if (status == ok)
      get_data(obj);          // e.g. (user-supplied; example above)
}
```

Non-blocking example (this uses a context and a callback function):

```
struct GetContext {
    WosOID oid;
};

// forward decl:
void get_cb(WosStatus status, WosObjPtr obj, void* pctx);

void get_object(WosClusterPtr wos, WosOID oid)
{
    GetContext* ctx = new GetContext();
    ctx->oid = oid;
    wos->Get(oid, get_cb, ctx);
}

void get_cb(WosStatus status, WosObjPtr obj, void* pctx)
{
    GetContext* ctx = static_cast<GetContext*>(pctx);

    if (status != ok) {
       printf("Error: Get %s failed %s\n", ctx->oid.c_str(), status.Er-
rMsg().c_str());
       delete ctx;
       return;
    }

    get_data(obj);           // e.g. (user-supplied: example above)
    delete ctx;
}
```

1.5.3 Delete

An object can be deleted using its `OID`.

Blocking example:

```
void del_image(WosClusterPtr wos, WosOID oid)
{
    WosStatus status;

    wos->Delete(status, oid);
    if (status == ok)
        update_database_remove_oid(oid); // e.g. (user-supplied)
}
```

Non-blocking example:

```
struct DelContext {
    WosOID oid;
};

// forward decl
void del_cb(WosStatus status, WosObjPtr obj, void* ctx);

void del_image(WosClusterPtr wos, WosOID oid)
{
    DelContext* ctx = new DelContext();
    ctx->oid = oid;

    wos->Delete(oid, del_cb, ctx);
}

void del_cb(WosStatus status, WosObjPtr obj, void* c)
{
    DelContext* ctx = static_cast<DelContext*>(c);

    if (status != ok) {
        printf("Delete failed on %s\n", ctx->oid.c_str());
        delete ctx;
        return;
    }

    // Update database indicating object no longer exists.
    //update_database_remove_oid(ctx->oid); // e.g.(user-supplied)

    delete ctx;
}
```

Notes:

- The `WosObjPtr` in the `Delete` callback is always null.
- If not all replicas of the object to be deleted are currently accessible to the client (for example, if there is a network failure to one zone), then the `Delete` operation will not be successful and a TemporarilyNotSupported error is returned. If a `Delete` fails, the object should be considered to be in an indeterminate state. It may still be retrievable or it may be actually be completely deleted. The `Delete` should be re-issued by the application.

1.5.4 Exists

An object can be checked for existence using its `OID`. A successful response conveys a high level of confidence that the object data would be returned on `Get()`, as `Exists()` ensures that WOS is able to locate all portions of the referenced object. This call does not read the entire object off of disk, thus, there is a possibility that WOS will need to attempt correction of data integrity issues at the time the object is actually retrieved.

Blocking example:

```
void exists_image(WosClusterPtr wos, WosOID oid)
{
   WosStatus status;

   wos->Exists(status, oid);
   if (status != ok) {
     printf("Error during Exists: %s\n", status.ErrMsg().c_str());
       return;
   }
   printf("Object Exists \n");
 }
```

Non-blocking example:

```
struct ExistsContext {
   WosOID oid;
};

// forward decl
void exists_cb(WosStatus status, WosObjPtr obj, void* ctx);

void exists_image(WosClusterPtr wos, WosOID oid)
{
   ExistsContext* ctx = new ExistsContext();
   ctx->oid = oid;

   wos->Exists(oid, exists_cb, ctx);
}

void exists_cb(WosStatus status, WosObjPtr obj, void* c)
{
   ExistsContext* ctx = static_cast<ExistsContext*>(c);

   if (status != ok) {
     printf("Error during Exists: %s\n", status.ErrMsg().c_str());
      delete ctx;
      return;
   }

   printf("Object Exists \n");
   delete ctx;
}
```

Notes:

The `WosObjPtr` in the `Exists` callback is not valid object and should not be used.

1.5.5 Reserve and PutOID

The `Reserve` and `PutOID` calls are to be used as a pair; they split the functionality normally executed by a single `Put` command into two commands, such that the commands can be executed at different points in time, or by different entities. Note that it is possible to send the reserved `OID` from one process to another and have the receiving process perform the `PutOID`.

Blocking example of `Reserve` and `PutOID`:

```
#define P_IMAGES "image"

// forward decl:
WosOID resv_oid(WosClusterPtr wos);
WosStatus put_obj(WosClusterPtr wos, WosOID oid, WosObjPtr obj);

void reserve_and_put(WosClusterPtr wos)
{
   WosOID oid = resv_oid(wos);                  // e.g. (see below)

   ...
   // perhaps a lot of things happen here.
   ...

   WosObjPtr obj = something_that_returns_an_object(); // e.g.

   WosStatus status = put_obj(wos, oid, obj);   // e.g. (see below)
}

WosOID resv_oid(WosClusterPtr wos)
{
   WosPolicy policy = wos->GetPolicy(P_IMAGES);
   WosStatus rstatus;            // return status
   WosOID roid;                  // return oid

   wos->Reserve(rstatus, roid, policy);

   return roid;
}

WosStatus put_obj(WosClusterPtr wos, WosOID oid, WosObjPtr obj)
{
   WosStatus status;

   wos->PutOID(status, oid, obj);

   if (status != ok) {
      printf("PutOID failed on oid %s\n", oid.c_str());
   }
   return status;
}
```

This example can also be re-written using the callback style API. Note that each callback triggers the next action in the chain.

Non-blocking example of `Reserve` and `PutOID`:

```
struct RsrvContext {
   int id;
};

struct PutOIDContext {
   int id;
   WosOID oid;
   char* data;
};

// forward decl:
void rsrv_cb(WosStatus status, WosObjPtr obj, void* pctx);
void putoid_cb(WosStatus status, WosObjPtr obj, void* pctx);

void step1(WosClusterPtr wos, WosPolicy policy, int id)
{
   RsrvContext* ctx = new RsrvContext();
   ctx->id = id;

   wos->Reserve(policy, rsrv_cb, ctx);
}

void rsrv_cb(WosStatus status, WosObjPtr obj, void* pctx)
{
   RsrvContext* ctx = static_cast<RsrvContext*>(pctx);

   if (status != ok) {
      printf("Reserve failed\n");
      delete ctx;
      return;
   }

   WosOID oid = obj->GetOID();
   // save OID for later use... when calling step2
   // update_database_reserve(id, oid);   // e.g. (user-supplied)

   delete ctx;
}

// Note that step2 could be called much later than the rsrv_cb,
//   and possibly in a different thread or process...
void step2(WosClusterPtr wos, WosOID oid, int id,
   char* data, size_t len, const char* type)
{
   // Create an object:
   WosObjPtr obj = create_object(data, len, type);    // e.g.

   PutOIDContext* ctx = new PutOIDContext();
   ctx->id = id;
   ctx->oid = oid;
   ctx->data = data;

   wos->PutOID(obj, oid, putoid_cb, ctx);
}

void putoid_cb(WosStatus status, WosObjPtr obj, void* pctx)
{
```

```
    PutOIDContext* ctx = static_cast<PutOIDContext*>(pctx);

    if (status != ok) {
       printf("PutOID failed: %s\n", status.ErrMsg().c_str());
       delete ctx;
       return;
    }

    // it is now possible to delete/free the memory associated
    // with the object that put referred to:
    // delete ctx->data;                // e.g. (user-supplied)

    // At this point, the put has succeeded. Update a database?
    // update_database_putoid(ctx->id, ctx->oid);

    delete ctx;
}
```

Notes:

- If not all replicas of the requested reservation are currently accessible to the client (for example, if there is a network failure to one zone), then the `PutOID` operation will not be successful and a TemporarilyNotSupported error is returned. If a `PutOID` fails, the `PutOID` should be re-issued by the application.

1.6 Streaming C++ API

The streaming API allows reading and writing of objects which would not fit in memory all at once. New methods have been added to the `WosCluster` class which allow the creation of a `WosGetStream` and a `WosPutStream`.

As with other critical objects in the API, all streams are accessed via a typedef'd boost `shared_ptr` as `WosGetStreamPtr`, and `WosPutStreamPtr`.

Errors that are detectable at point-of-invocation (in other words, method call) are indicated by throwing exceptions. Errors associated with a result, or delivered in a callback routine are indicated via the `WosStatus` return parameter.

In general, it is important to check the value of the `WosStatus` return parameter before inspecting any returned `WosObjPtr`.

1.6.1 PutStreams

A `PutStream` is created from the WosCluster's `CreatePutStream` call. `PutStream`s allow writing to an object in successive chunks, or *spans*. A span has an offset (past the beginning of the object) and a length.

When writing to the `PutStream` is complete, the `PutStream`'s `Close()` method must be called to retrieve the associated OID.

Notes:

- There is a blocking and a non-blocking (callback-style) version of the `PutSpan` and `Close` methods. Overall, higher throughput can be achieved by using the non-blocking style calls.
- `PutStream`s are not thread-safe. The behavior if you call `PutSpan` on the same stream simultaneously in different threads is undefined. However, different threads can concurrently operate on unique `PutStream`s.
- It is acceptable to specify `PutSpan`s which overlap. However, WOS assumes that any overlapping data will be identical. If data is written to the `PutStream`, and then written again with different data, the result is undefined.
- It is acceptable to not specify every byte of data in a `PutStream` (in other words, leave "gaps" between `PutSpan`s that are written). Any unspecified bytes will be returned by a `GET` operation as 0s.

Blocking example:

```
WosOID
BigPut(WosClusterPtr wos, WosPolicy policy,
    char* data[], int n, int eachlen)
{
    // We assume for this example that 'data' contains
    //   'n' pointers to blocks of length 'eachlen'.
    WosPutStreamPtr ps = wos->CreatePutStream(policy);
```

```
    ps->SetMeta("key", "important");

    uint64_t off = 0;
    WosStatus rstatus;

    for (int i = 0; i < n; i++) {
       ps->PutSpan(rstatus, data[i], off, eachlen);
       off += eachlen;

       if (rstatus != ok) {
          printf("Error in PutSpan %d\n", i);
       }
    }

    WosOID roid;
    ps->Close(rstatus, roid);

    if (rstatus != ok)
       printf("Error during PutStream Close\n");

    return roid;
}
```

The non-blocking example is similar, but additionally has a callback function (`PutSpan_cb`), and a call to `Wait()`, (which assures that all non-blocking functions submitted so far have completed). In some cases, it may be useful to use the non-blocking form of `Close()` as well.

When using the non-blocking `PutSpan()` method, the data (referenced by a pointer) passed to `PutSpan` must remain valid until the corresponding callback is called.

Note further that each invocation of a callback will pass a user-defined "context" pointer to the callback on completion. This context may be a pointer to any user-specified piece of data. Since callbacks may occur in any order, this technique allows coordination of data between the caller and the callback. If contexts are used, it is important to arrange for their deletion in callback functions. It is permissible to pass a null pointer(0) as the context if this feature is not desired.

Non-blocking example:

```
// BigPut_nb shows how to use non-blocking PutSpans to write
//   a large object.
// Contexts (user-defined structs with user data) are created
//   to transfer information (context) between the calls to
//   PutSpan and their respective callbacks in PutSpan_cb.
struct PutSpanContext {
   int n;
   const char* data;

   PutSpanContext(int x, const char* p) : n(x), data(p) {}
};

void
PutSpan_cb(WosStatus s, WosObjPtr, WosCluster::Context c)
{
   PutSpanContext* psc = static_cast<PutSpanContext*>(c);

   if (s != ok)
      printf("Error in Span %d\n", psc->n);

   // possibly de-allocate/free/delete data:
   // delete psc->data;

   delete psc;
}

WosOID
BigPut_nb(WosClusterPtr wos, WosPolicy policy,
   char* data[], int n, int eachlen)
{
   // We assume for this example that 'data' contains
   //   'n' pointers to blocks of length 'eachlen'.
   WosPutStreamPtr ps = wos->CreatePutStream(policy);
   ps->SetMeta("Key", "important");

   uint64_t off = 0;

   for (int i = 0; i < n; i++) {
      PutSpanContext* ctx = new PutSpanContext(i, data[i]);
      ps->PutSpan(data[i], off, eachlen, ctx, PutSpan_cb);
      off += eachlen;
   }

   wos->Wait();

   WosStatus rstatus;
   WosOID roid;
   ps->Close(rstatus, roid);

   if (rstatus != ok)
      printf("Error during PutStream Close\n");

   return roid;
}
```

1.6.2 PutOIDStreams

`PutOIDStreams` allow using a PutStream with a pre-reserved OID (obtained from a previous call to `Reserve()`). After creation of the stream, the rest of the calls are identical in behavior to PutStreams. In fact, note that `CreatePutOIDStream` returns a `WosPutStreamPtr` (not a `WosPutOIDStreamPtr`).

Example:

```
void
BigPutOID(WosClusterPtr wos, WosOID oid,
    Char* data[], int n, int eachlen)
{
    WosPutStreamPtr ps = wos->CreatePutOIDStream(oid);
    // …

    WosStatus rstatus;
    WosOID roid;

    ps->Close(rstatus, roid);
}
```

1.6.3 GetStream

A `GetStream` is used to access large objects piece-by-piece, or in *spans*. Like a `PutStream`, it contains both a blocking and non-blocking form of the `GetSpan()` method.

Blocking example:

```
// This example retrieves a large object from WOS, and write
//   it to a file piece by piece.
void GetBig(WosClusterPtr wos, WosOID oid, const char* filename)
{
   int fd = open(filename, O_RDWR | O_CREAT, 0644);

   try {
      WosGetStreamPtr gets = wos->CreateGetStream(oid);
   }
   catch (WosE_ObjectNotFound& e) {
      Printf("Invalid OID: %s\n", oid.c_str();
      Return;
   }

   uint64_t len = gets->GetLength();
   uint64_t chunksize = 32768;

   uint64_t off = 0;
   WosStatus rstatus;            // return status
   WosObjPtr robj;               // return object for the span
   while (len > 0) {
      // only "take" up to chunksize bytes:
      uint64_t take = len > chunksize ? chunksize : len;
      len -= take;
      off += take;

      gets->GetSpan(rstatus, robj, off, take);
      if (rstatus == ok) {
         const void* p;
         uint64_t objlen;
         robj->GetData(p, objlen);

         write(fd, p, objlen);
      }
   }

   close(fd);

   // implicitly release GetStream when 'gets' goes out-of-scope
}
```

Higher throughput may be achieved using the non-blocking form of GetSpan (since more work can be executed ahead, in parallel, in the cluster). Note, however, that non-blocking GetSpan callbacks may occur in any order, so some care may be necessary in some applications.

Non-blocking example:

```
// GetBig_nb transfers an object from the cluster to a
//   local file, writing 32kB "chunks" in the order
//   they're received (which may not be strictly sequentially).
```

```
struct GetSpanContext {
   int fd;
   uint64_t off;
   uint64_t len;

   GetSpanContext(int _fd, uint64_t _off, uint64_t _len)
      : fd(_fd), off(_off), len(_len) {}
};

void GetBig_cb(WosStatus s, WosObjPtr obj, Context c)
{
   GetSpanContext* ctx = static_cast<GetSpanContext*>(c);

   if (s == ok) {
      lseek(ctx->fd, ctx->off, SEEK_SET);

      const void* p;
      uint64_t objlen;
      obj->GetData(p, objlen);

      write(ctx->fd, p, objlen);
   }

   delete ctx;
}

void GetBig_nb(WosClusterPtr wos, WosOID oid, const char* filename)
{
   int fd = open(filename, O_RDWR | O_CREAT, 0644);

   try {
      WosGetStreamPtr gets = wos->CreateGetStream(oid);
   }
   catch (WosE_ObjectNotFound& e) {
      Printf("Invalid OID: %s\n", oid.c_str();
      Return;
   }

   uint64_t len = gets->GetLength();
   uint64_t chunksize = 32768;

   uint64_t off = 0;
   while (len > 0) {
      // only "take" up to chunksize bytes:
      uint64_t take = len > chunksize ? chunksize : len;
      len -= take;
      off += take;

      GetSpanContext* ctx = new GetSpanContext(fd, off, take);
      gets->GetSpan(off, take, ctx, GetBig_cb);
   }

   // Wait for all Spans to complete
   wos->Wait();
}
```

1.6.4 Optimization of GetSpan for Small Subobject Readback

`GetStreams` are optimized for readback of large objects. However, if your use case calls for reading back small parts of objects as quickly as possible then the following advanced options may be used.

There are two advanced options to `GetSpan` in addition to those described in the previous section. These are the full prototypes of the blocking and non-blocking `GetSpan` methods.

Blocking:

```
void GetSpan(WosStatus& status, WosObjPtr& obj,
    uint64_t off, uint64_t len,
    BufferMode mode = Buffered,
    IntegrityCheck integrity_check = IntegrityCheckEnabled)
```

Non-Blocking:

```
void GetSpan(uint64_t off, uint64_t len, Context context,
    Callback cb,
    BufferMode mode = Buffered,
    IntegrityCheck integrity_check = IntegrityCheckEnabled)
```

1.6.5 Buffer Mode

Internal to WOS, objects greater than 1 MB are broken up into 1 MB pieces. By default, when you read back a span of data, all 1 MB pieces that overlap the span are read into the memory space of the client and are cached there for a short while. It is often the case the next span requested is subsequent to the previous one, and so it will use the remainder of the previously cached 1 MB piece of the object. This allows an efficient readback of a large object, span by span. This is the default behavior, which is called `Buffered` mode.

An `Unbuffered` mode is available, which is optimized for random access of small parts of large objects. In Unbuffered mode, only the data requested by the client is read back from WOS, not the entire 1 MB chunk that contains the data. The result is less network traffic. However, there is no caching of previously read back data, so this mode should not be used when doing a serial readback of a large object.

1.6.6 Integrity Check

All WOS objects are written to disk along with a checksum. When the object is accessed, the object data is compared with the checksum to verify that the object is intact. In the event that the object has been corrupted, WOS self-heals by replacing this corrupt replica with a valid replica from elsewhere in the cluster.

This integrity check can be disabled by setting the `IntegrityCheck` parameter of `GetSpan` to `IntegrityCheckDisabled`. This is a performance optimization when reading back small spans of larger objects, because the integrity check operation requires the entire 1 MB piece to be read from disk and verified against its checksum. Disabling the integrity check allows WOS to read back only the data requested by the client.

Note, however, that disabling the integrity check not only prevents WOS from identifying the requested data as corrupted, but it also bypasses the self-healing function. If you disable this integrity check, it is important that you have a way of identifying data corruption in your own application, and in the case where data is corrupted you must repeat the `GetSpan` call with `IntegrityCheckEnabled` to trigger the self-healing function.

NOTE : The combination of Buffered and IntegrityCheckDisabled is not allowed.

1.6.7 GetStreams and Metadata

GetStreams support a number of metadata-specific features that are not available in the whole-object API's:

- It is possible to get an object without fetching its metadata.
- It is possible to retrieve an object's metadata without fetching its data.

When opening a GetStream, there is an optional prefetch_meta Boolean parameter (which defaults to true), that permits you to avoid retrieving metadata (unless it is subsequently referenced, specifically, via a GetMeta call).

Example:

```
void MetaNoPrefetch(WosClusterPtr wos, WosOID oid)
{
   bool prefetch_meta = false;
   WosGetStreamPtr gets = wos->CreateGetStream(oid, prefetch_meta);
   // . . .
}
```

Since creating a GetStream does not initiate retrieving object data (any of its spans) automatically, it is possible to access an object's metadata without retrieving data (by simply not calling GetSpan):

Example:

```
// MetaDataOnly is a code snippet that shows how to examine
//   all of the metadata for an object without retrieving the
//   object itself.
using namespace std;
void MetaFunc(Context c, string key, string value)
{
   printf("Meta key=%s value=%s\n", key.c_str(), value.c_str());
}

void MetaDataOnly(WosClusterPtr wos, WosOID oid)
{
   WosGetStreamPtr gets = wos->CreateGetStream(oid);

   gets->EachMeta(0, metafunc);// context is not used.
}
```

1.6.8 Multi-part Upload

Refer to Appendix A for description of the multi-part upload feature.

To construct a multi-part PutStream

```
WosPutStreamPtr CreateMultiPartPutStream(WosPolicy pol);
```

This method constructs a **`WosPutStream`** that is capable of accepting addition of part objects.

To add parts

Callback-style API:

```
void PutPart(WosOID oid,  Context context, Callback cb);
```

This method on a PutStream appends the next part to the multi-part object. Parts are appended corresponding to the order of the PutPart calls.

Blocking API:

```
void PutPart(WosStatus& status, WosOID oid);
```

NOTE : Calling PutSpan on a stream opened with CreateMultiPartPutStream will result in a WosE_NotImpl exception being thrown.

Example:

```
/*
This sample uses a multipart stream to create an object assembled from
other parts.
Sample output:

Reading back object parts:
Part 0: oid = wAx-CkQ7DgxEhFBA4xL1B613HD0Sxthg_lgfPxlI, data = 'Hello'
Part 1: oid = PAOB-XfEDfN3u6BMqoCGQQajkAZghjKGawm0K_8B, data = ' '
Part 2: oid = zAy9OUQ4Dj90hGBNRaS2DGMhHDYOZ8-fih3pLOxI, data = 'World'
Part 3: oid = YCZWgu6TBITOLtDsJZJSHSDL0JgPPn3sHZfIA20L, data = '!'
Reading back multipart object:
Hello World!
*/

#include <iostream>
#include <string>
#include <vector>

#include <wos_cluster.hpp>

using wosapi::WosCluster;
using wosapi::WosClusterPtr;
using wosapi::WosOID;
using wosapi::WosObj;
using wosapi::WosObjPtr;
using wosapi::WosPolicy;
using wosapi::WosPutStreamPtr;
using wosapi::WosStatus;
```

```
// Given a vector of strings, stores each string as a WOS object.
// Returns the corresponding OIDs in a vector.
std::vector<WosOID> CreateObjectParts(const WosClusterPtr& wos,
                                      const WosPolicy& policy,
                                  const std::vector<std::string>& parts)
{
    std::vector<WosOID> oids;

    for (std::vector<std::string>::const_iterator it = parts.begin();
            it != parts.end(); ++it)
    {
        WosObjPtr obj = WosObj::Create();
        obj->SetData(it->c_str(), it->size());

        WosStatus status;
        WosOID oid;
        wos->Put(status, oid, policy, obj);
        oids.push_back(oid);
    }

    return oids;
}

// Given a list of OIDs, stores each OID in a multipart stream to
// assemble a larger object. Returns the multipart OID.
WosOID CreateMultipartObject(const WosClusterPtr& wos,
                             const WosPolicy& policy,
                             const std::vector<WosOID> oids)
{
    WosPutStreamPtr puts = wos->CreateMultiPartPutStream(policy);

    for (std::vector<WosOID>::const_iterator it = oids.begin();
            it != oids.end(); ++it)
    {
        WosStatus status;
        puts->PutPart(status, *it);
    }

    WosStatus status;
    WosOID moid;
    puts->Close(status, moid);

    return moid;
}
int main()
{
    std::string ip_addr = "wos7k12.sol";
    std::string policy = "single";

    WosClusterPtr wos = WosCluster::Connect(ip_addr);
    WosPolicy wos_policy = wos->GetPolicy(policy);

    std::string part_data[] = {"Hello", " ", "World", "!"};
    std::vector<std::string> part_data_vec(part_data, part_data +
sizeof(part_data) / sizeof(std::string));

    // create object parts
    std::vector<WosOID> parts = CreateObjectParts(wos, wos_policy,
part_data_vec);
```

```
    std::cout << "Reading back object parts:" << std::endl;
    for (size_t i = 0; i < parts.size(); ++i)
    {
        WosStatus status;
        WosObjPtr obj;
        wos->Get(status, parts[i], obj);

        const void* data; uint64_t len;
        obj->GetData(data, len);

        std::cout << "  Part " << i << ": oid = " << parts[i] <<
           ", data = '" << std::string(static_cast<const char*>(data)
, len)  <<
            "'" << std::endl;
    }

    // create multipart object from parts
    WosOID moid = CreateMultipartObject(wos, wos_policy, parts);

    std::cout << std::endl << "Reading back multipart object:" <<
std::endl;
    WosStatus status;
    WosObjPtr obj;
    wos->Get(status, moid, obj);
    const void* data; uint64_t len;
    obj->GetData(data, len);
    std::cout << "  " << std::string(static_cast<const char*>(data),
len) << std::endl;

    return EXIT_SUCCESS;
}
```

1.6.9 Stream Index Caching

Stream index caching can be used to maintain an internal cache of the index objects to avoid costly disk I/O for frequently accessed large objects.

Invoking `SetStreamCacheSize(size_t cache_size)` with the size argument greater than 0 (zero) will enable stream cache. Specifying 0 (zero) disables the stream cache. By default, the stream cache is disabled.

The stream cache is finite in size and employs LRU (least recently used) policy for eviction. When the stream cache becomes full, the oldest item gets evicted from the cache in order to make way for the newest addition. Data can remain in caches after deletes until evicted.

1.7 Exceptions

Some errors are delivered as C++ exceptions:

- All of the non-blocking API calls (those that take a Callback and Context as parameters) may throw a `WosE_MissingCallback` if a "null" callback is passed in.
- When performing a "`Get`" on a large object, if it is determined that the object will not fit in memory, a `WosE_CannotAllocate` exception will be thrown. This does not apply to streaming APIs.
- All WOS exceptions are declared in "`wos_exception.hpp`" and derive from the WosException class (which, in turn is derived from std::exception), thus, it is possible to catch either a specific WOS exception, or all WOS exceptions, or any combination of the two.

Example:

```
#include <wos_cluster.hpp>

void ExceptionExample(WosClusterPtr wos, WosOID oid)
{
    WosStatus rstatus;// return status
    WosObjPtr robj;// return object ptr
    try {
        wos->Get(rstatus, oid, robj);
    }
    catch (WosE_CannotAllocate& e) {
        printf("Cannot allocate object\n");
    }
    // catch all other WOS exceptions:
    catch (WosException& e) {
        printf ("Error: %s\n", e.what());
    }
}
```

`WosCluster::Connect` may throw the following exception:

- `WosE_CannotConnect` — Thrown if application cannot connect to the cluster.

`CreateGetStream` may throw the following exception:

- `WosE_ObjectNotFound` — A OID was specified that refers to a non-existent object.

`CreatePutStream` may throw the following exception:

- `WosE_InvalidPolicy` — A policy name was specified that is not defined in the cluster.

`CreatePutOIDStream` may throw the following exception:

- `WosE_InvalidReservation` — A OID was specified that was not obtained via a call to `Reserve`, or that has already been written.

PutSpan may throw the following exceptions:

- WosE_MissingCallback — A non-blocking PutSpan was called without specifying a Callback routine.
- WosE_StreamFrozen — PutSpans cannot be specified after a PutStream is closed.
- WosE_EmptyStream — PutSpan was called with a 0-byte span.

Close may throw the following exception:

- WosE_StreamFrozen — A closed PutStream cannot be re-closed.
- WosE_EmptyStream — The closed PutStream has no data nor metadata.

SetMeta may throw the following exceptions:

- WosE_MaxMetaCount — Each object supports a maximum of 32000 pieces of metadata. A maximum of 64MB of metadata is supported.
- WosE_InvalidKeyName — Metadata keys must not start with ":".
- WosE_StreamFrozen — PutSpans cannot be specified after a PutStream is closed.

The blocking GetMeta may throw the following exception:

- WosE_ObjectNotFound — An internal error prevents access to this object's metadata.

GetSpan may throw the following exceptions:

- WosE_InvalidOffset — Offset specified is < 0 or > length of object.
- WosE_InvalidLength — Length specified is < 0 or > length of object.
- WosE_CannotAllocate — A request for a span was made which will not fit in memory.
- WosE_InvalidGetSpanMode — An invalid combination of BufferMode and IntegrityCheck was specified. (The combination of Buffered and IntegrityCheckDisabled is not allowed.)

Chapter 2
Java API

2.1 Introduction

The Java API provides a Java interface to all WOS operations exposed by the C++ API, including:

- Connect to cluster
- Create WOS Objects; assign/examine object data and metadata
- `PUT`, `GET`, `DELETE`, `EXISTS` object from cluster
- `Reserve`, `PutOID`

The Java API is supported on RHEL6 64-bit Linux only; Java versions J2SE 1.4.2 and above.

NOTE : GOA is only accessible via the RESTful HTTP API.

2.2 Installation

There are three WOS libraries required to use the Java API: `libwos_cpp.so` (the underlying C++ client library), `libwosjava.so` (Java bridge to `libwos_cpp`), and `wosjava.jar` (the Java API itself).

The shared libraries may be installed at the system level (i.e. in `/usr/lib64/...`) or be installed locally to a user.

If the libraries are installed locally, one must manually modify the dynamic loader search path (using `export LD_LIBRARY_PATH=...`).

The Java library (`wosjava.jar`) must be in the application's classpath.

2.3 Connecting to the Cluster

Before performing WOS operations, your application must connect to the cluster. Connect is a somewhat expensive operation; it is intended that applications do this once (or infrequently) and then keep their connection open while performing WOS operations on the cluster. Only one connection to the cluster per process may be open at one time.

Example:

```
import com.ddn.wos.api.*;

public class Demo {
    public static void main(String[] args) {
        String host = "10.0.0.2";

        try {
            WosCluster wos = new WosCluster();
            wos.connect(host);
        }
        catch (WosException e) {
            System.out.println("WosException: " + e.getMessage());
        }
    }
}
```

Notes:

- An application can connect to the cluster by contacting any node in the cluster. It is recommended that a DNS entry be created which round-robins through the individual node addresses. This will enable retry logic inside of “connect” to connect to the cluster when some nodes are inaccessible.

2.4 Creating/Examining WosObjects

Prior to storing data to the WOS cluster, data and its associated metadata must be bound with a `Java WosObject`. Data is described as a byte array, while metadata is described by (key, value) pairs.

Example:

```
public WosObj createObject(byte[] data, String type) {

    WosObj obj = new WosObj();

    obj.setData(data);
    obj.setMeta("Content-Type", type.getBytes());

    return obj;
}
```

When data is retrieved from the cluster, it is returned in the form of a WosObj, from which you can extract the data and metadata. The most common way to retrieve metadata is to request the data for a known key (obj.getMeta(key)). Alternatively, a map containing all metadata for an object is available through obj.getMetaAll().

Example:

```
import java.util.Map;
import java.util.Iterator;

public void showData(WosObj obj) {
    byte[] data = obj.getData();
    String contentType = new String(obj.getMeta("Content-Type"));

    Map metaMap = obj.getMetaAll();
    Iterator itr = metaMap.entrySet().iterator();
    while (itr.hasNext()) {
        Map.Entry entry = ((Map.Entry)itr.next());
        String key = (String)entry.getKey();
        byte[] value = (byte[])entry.getValue();
    }
}
```

2.5 Standard WOS Operations

The standard WOS operations are `Put`, `Get`, `Delete`, `Exists`, `Reserve`, and `PutOID`. For more information on these operations, see Section 1.5. This section describes the Java interface to these operations.

The Java API supports blocking and non-blocking (callback-style) operations. When a blocking operation is issued, the calling thread blocks until the operation completes. When a non-blocking operation is issued, control is immediately returned to the calling thread while the WOS operation executes in the background. When the operation is complete, a user-specified callback is executed. Using this non-blocking technique, many WOS operations can be run in parallel.

To maximize WOS performance:

- For single-threaded applications, use the non-blocking (callback-style) operations.
- For multi-threaded applications, use the blocking operations (potentially running many in parallel).

2.5.1 Put

Blocking Example:

```
public void writeObject(WosCluster wos, byte[] data, String policy, String
userId) {
    try {
        WosObj obj = new WosObj(data);
        obj.setMeta("key1", "value1".getBytes());
        obj.setMeta("key2", "value2".getBytes());

        String oid = wos.put(obj, policy);

        // The object has been successfully stored.
        // You have a valid oid. For example:
        // updateDatabase(oid, userId);
    }
    catch (WosException e) {
        System.out.println("Error during Put: " + e.getMessage());
    }
}
```

Non-Blocking Example:

```
public void writeObjectNB(WosCluster wos, byte[] data, String policy, String
userId) {
    WosObj obj = new WosObj(data);
    obj.setMeta("key1", "value1".getBytes());
    obj.setMeta("key2", "value2".getBytes());

    // Issue Put command; the PutCB object stores an identifier
    // (or whatever) related to the request, which can be used
    // later by the callback function. In this case, I pass
    // the userId, which is stored in the database along with
    // the resulting OID.
    wos.put(obj, policy, new PutCB(userId));
}

private static class PutCB implements WosPutCB {
    private String userId;
```

```
    public PutCB(String userId) {
        this.userId = userId;
    }
    public void cb(WosStatus status, String oid) {
        if (!status.equals(WosStatus.OK)) {
            System.out.println("Error during Put: " + status.getMessage());
            return;
        }
        // The object has been successfully stored.
        // You have a valid oid. For example:
        // updateDatabase(oid, userId);
    }
}
```

Notes:

- Once a `WosObject` has been submitted for `Put` to the cluster, it becomes *frozen*; after which, no further changes to its data or metadata are permitted.
- The callback presents a valid OID only when the `WosStatus` indicates that the `Put` operation succeeded.

2.5.2 Get

An object can be retrieved from the cluster using its `OID`.

Blocking Example:

```
public void getImage(WosCluster wos, String oid) {
    try {
        WosObj obj = wos.get(oid);
        byte[] imagedata = obj.getData();
    }
    catch (WosException e) {
        System.out.println("Error during Get: " + e.getMessage());
    }
}
```

Non-Blocking Example:

```
public void getImageNB(WosCluster wos, String oid) {
    // Issue Get command
    wos.get(oid, new GetCB());
}

private static class GetCB implements WosGetCB {
    public void cb(WosStatus status, WosObj obj) {
        if (!status.equals(WosStatus.OK)) {
            System.out.println("Error during Get: " + status.getMessage());
            return;
        }
        byte[] imagedata = obj.getData();
    }
}
```

Notes:

- The callback presents a valid `WosObj` only when the `WosStatus` indicates that the `Put` operation succeeded.

2.5.3 Delete

An object can be deleted using its `OID`.

Blocking Example:

```
public void delObject(WosCluster wos, String oid, String userId) {
    try {
        wos.delete(oid);

        // The object has been successfully removed from storage.
        // updateDatabase(oid, userId);
    }
    catch (WosException e) {
        System.out.println("Error during delete: " + e.getMessage());
    }
}
```

Non-Blocking Example:

```
public void delObjectNB(WosCluster wos, String oid, String userId) {
    // Issue Delete command
    wos.delete(oid, new DeleteCB(oid, userId));
}

private static class DeleteCB implements WosDeleteCB {
    private String oid;
    private String userId;
    public DeleteCB(String oid, String userId) {
        this.oid = oid;
        this.userId = userId;
    }
    public void cb(WosStatus status) {
        if (status.equals(WosStatus.OK)) {
            // The object has been successfully removed from storage.
            // updateDatabase(oid, userId);
        }
        else {
            System.out.println("Error during delete: " +
status.getMessage());
        }
    }
}
```

Notes:

- If not all replicas of the object to be deleted are currently accessible to the client (for example, if there is a network failure to one zone), then the `Delete` operation will not be successful and a TemporarilyNotSupported error is returned. If a `Delete` fails, the object should be considered to be in an indeterminate state. It may still be retrievable or it may be actually be completely deleted. The `Delete` should be re-issued by the application.

2.5.4 Exists

An object can be checked for existence using its `OID`. A successful response conveys a high level of confidence that the object data would be returned on `Get()`, as `Exists()` ensures that WOS is able to locate all portions of the referenced object. This call does not read the entire object off of disk, thus, there is a possibility that WOS will need to attempt correction of data integrity issues at the time the object is actually retrieved.

Blocking Example:

```
public void existsObject(WosCluster wos, String oid, String userId) {
   try {
      if (wos.exists(oid)) {
         // object exists
      }
      else {
         // object does not exists
      }
   }
   catch (WosException e) {
      System.out.println("Error during exists: " + e.getMessage());
   }
}
```

Non-Blocking Example:

```
public void existsObjectNB(WosCluster wos, String oid) {
   // Issue Exists command
   wos.exists(oid, new ExistsCB(oid));
}

private static class ExistsCB implements WosExistsCB {
   private String oid;
   public ExistsCB(String oid) {
      this.oid = oid;
   }
   public void cb(WosStatus status) {
      if (status.equals(WosStatus.OK)) {
         // The object exists.
      }
      else {
         // object does not exists
         System.out.println("Error during exists: " +
               status.getMessage());
      }
   }
}
```

2.5.5 Reserve and PutOID

The `Reserve` and `PutOID` calls are to be used as a pair. They split the functionality normally executed by a single `Put` command into two commands, such that the commands can be executed at different points in time, or by different entities.

Blocking Example:

```
public void reservePut(WosCluster wos, String policy) {
    try {
        String oid = wos.reserve(policy);
        ...
        // Other things may happen here
        ...
        byte[] data = get_user_data();
        WosObj obj = new WosObj(data);
        wos.putoid(obj, oid);
    }
    catch (WosException e) {
        System.out.println("Error during reserve/putoid: " +
e.getMessage());
    }
}
```

Non-Blocking Example:

```
public void step1(WosCluster wos, int id, String policy) {
    // Issue reserve
    wos.reserve(policy, new ReserveCB(id));
}

private static class ReserveCB implements WosReserveCB {
    int id;
    public ReserveCB(int id) {
        this.id = id;
    }
    public void cb(WosStatus status, String oid) {
        if (!status.equals(WosStatus.OK)) {
            System.out.println("Error reserving object" +
status.getMessage());
            return;
        }

        // Save OID for later use
        // updateDatabase(id, oid); // e.g. (user-supplied)
    }
}

// Note that step2 could be called much later than ReserveCB.cb(),
//     and possibly in a different thread or process...
public void step2(WosCluster wos, String oid, int id, byte[] data) {
    WosObj obj = new WosObj(data);
    wos.putoid(obj, oid, new PutOIDCB(id, oid));
}

private static class PutOIDCB implements WosPutOIDCB {
    int id;
    String oid;
    public PutOIDCB(int id, String oid) {
        this.id = id;
```

```
        this.oid = oid;
    }
    public void cb(WosStatus status) {
        if (!status.equals(WosStatus.OK)) {
            System.out.println("Error reserving object" +
status.getMessage());
            return;
        }

        // Put successful. Update database?
        // updateDatabasePutoid(id, oid);
        // e.g. (user-supplied)
    }
}
```

Notes:

- It is possible to send the reserved `OID` from one process to another and have the receiving process perform the `PutOID`.
- If not all replicas of the requested reservation are currently accessible to the client (for example, if there is a network failure to one zone), then the `PutOID` operation will not be successful and a TemporarilyNotSupported error is returned. If a `PutOID` fails, it should be re-issued by the application.

2.6 Streaming Java API

The streaming API allows reading and writing of objects which would not fit in memory all at once.

Errors that are detectable at point-of-invocation (i.e. method call) are indicated by throwing exceptions. Errors delivered in a callback routine are indicated via the `WosStatus` return parameter. In general, it is important to check the value of the `WosStatus` return parameter before inspecting any returned `WosObj`.

2.6.1 PutStreams

A `PutStream` is created from the WosCluster's `CreatePutStream` call. `PutStream`s allow writing to an object in successive chunks, or *spans*. A span is an offset (past the beginning of the object) and a length.

When writing to the `PutStream` is complete, the `PutStream`'s `Close()` method must be called to retrieve the associated `OID`.

Notes:

- There is a blocking and a non-blocking (callback-style) version of the `PutSpan` and `Close` methods. Overall, higher throughput can be achieved by using the non-blocking style calls.
- It is acceptable to specify `PutSpan`s which overlap. However, WOS assumes that any overlapping data will be identical. If data is written to the `PutStream`, and then written again with different data, the result is undefined.
- It is acceptable to not specify every byte of data in a `PutStream` (in other words, leave "gaps" between `PutSpan`s that are written). Any unspecified bytes will be returned by a `GET` operation as 0s.

Blocking Example:

```
public String bigPut(WosCluster wos, String policy) {
    try {
        // We assume that there is an external data source from
        // which we can read segments of the data.
        WosPutStream puts = wos.createPutStream(policy);

        long offset = 0;
        while (datasource.hasNext()) {
            byte[] data = datasource.next();
            puts.putSpan(data, offset, data.length);
            offset += data.length;
        }
        String oid = puts.close();
        return oid;
    }
    catch (WosException e) {
        System.out.println("Error writing object: " + e.getMessage());
    }
    return null;
}
```

The non-blocking example is similar, but additionally has a callback interface (WosPutSpanNB), and a call to woswait(), (which assures that all non-blocking functions submitted so far have completed). In some cases, it may be useful to use the non-blocking form of Close() as well.

Non-Blocking Example:

```
public String bigPutNB(WosCluster wos, String policy) {
   try {
      // We assume that there is an external data source from
      // which we can read segments of the data.
      WosPutStream puts = wos.createPutStream(policy);

      long offset = 0;
      while (datasource.hasNext()) {
         byte[] data = datasource.next();
         puts.putSpan(data, offset, new PutSpanCB(offset));
         offset += data.length;
      }
      wos.woswait();

      String oid = puts.close();
      return oid;
   }
   catch (WosException e) {
      System.out.println("Error writing object: " + e.getMessage());
   }
   return null;
}

private static class PutSpanCB implements WosPutSpanCB {
   long offset;
   public PutSpanCB(long offset) {
      this.offset = offset;
   }
   public void cb(WosStatus status) {
      if (!status.equals(WosStatus.OK)) {
         System.out.printf("Error writing object at offset %d: %s",
offset, status.getMessage());
      }
   }
}
```

2.6.2 PutOIDStreams

`PutOIDStreams` allow using a `PutStream` with a pre-reserved `OID` (obtained from a previous call to `Reserve()`). After creation of the stream, the rest of the calls are identical in behavior to `PutStream`s. In fact, note that `CreatePutOIDStream` returns a `WosPutStream` object.

Example:

```
public String bigPutoid(WosCluster wos, String policy) {
    try {
        String oid = wos.reserve(policy);

        // ...

        WosPutStream puts = wos.createPutStream(policy);

        // ...

        puts.close();
        return oid;
    }
    catch (WosException e) {
        System.out.println("Error writing object: " + e.getMessage());
    }
    return null;
}
```

2.6.3 CreateMultiPartPutStreams

Refer to Appendix A for description of the multi-part upload feature.

`CreateMultiPartPutStreams` allow using a `PutStream` with a pre-reserved `OID` (obtained from a previous call to `Reserve()`). After creation of the stream, the rest of the calls are identical in behavior to `PutStreams`. In fact, note that `CreateMultiPartPutStream` returns a `WosPutStream` object.

A `putPart` method on a `WosPutStream` is available for use only on streams opened with `CreateMultiPartPutStream`.

Example:

```
/*
    This sample uses a multipart stream to create an object assembled
    from other parts.

    Sample output:

    Reading back object parts:
      Part 0 : oid = NAMD23fGDdFXu4BAMHkXgqu6FAFIOI0-C-lCPL8P, data = 'Hello'
      Part 1 : oid = pAonm1WiD5VVncBhYP6EVrcTWCZ9n0nVfa2_PJUN, data = ' '
      Part 2 : oid = HBGJfTNMCXsz8yAEF1IDtDNDDE2oIQjrnRY6IQkA, data = 'World'
      Part 3 : oid = kBlqThFvC0gB0RAkipKYM0fQfGbsKmjqswxEO_1E, data = '!'

    Reading back multipart object:
      Hello World!
*/

import java.util.Arrays;
import java.util.List;
import java.util.Stack;

import com.ddn.wos.api.WosCluster;
import com.ddn.wos.api.WosException;
import com.ddn.wos.api.WosObj;
import com.ddn.wos.api.WosPutStream;

public class MultipartSample {
    // Given a list of strings, stores each string as a WOS object. Returns the
    // corresponding OIDs in a list.
   public static List<String> createObjectParts(WosCluster wos, String policy,
List<String> parts)
        throws WosException {
        Stack<String> oids = new Stack<String>();

        for (String part : parts) {
            WosObj obj = new WosObj(part.getBytes());
            String oid = wos.put(obj, policy);
            oids.push(oid);
        }

        return oids;
    }

   // Given a list of OIDs, stores each OID in a multipart stream to assemble a
    // larger object. Returns the multipart OID.
    public static String createMultipartObject(WosCluster wos, String policy,
List<String> oids)
        throws WosException {
        WosPutStream puts = wos.createMultiPartPutStream(policy);
```

```
            for (String oid : oids) {
                puts.putPart(oid);
            }

            return puts.close();
        }

        public static void main(String[] args) {
            final String ipAddr = "wos7k12.sol";
            final String policy = "single";

            try {
                WosCluster wos = new WosCluster();
                wos.connect(ipAddr);

                // create object parts
                List<String> parts = createObjectParts(wos, policy,
                        Arrays.asList(new String[] {"Hello", " ", "World", "!"}));

                System.out.println("Reading back object parts:");
                for (int i = 0; i < parts.size(); i++) {
                    String oid = parts.get(i);
                    WosObj obj = wos.get(oid);
                System.out.println(  "  Part " + i + " : oid = " + oid + ", data = '" +
                        new String(obj.getData()) + "'");
                }

                // create multipart object from parts
                final String moid = createMultipartObject(wos, policy, parts);

                System.out.println("\nReading back multipart object:");
                WosObj obj = wos.get(moid);
                System.out.println("  " + new String(obj.getData()));
            } catch (WosException e) {
            }
        }
    }
```

2.6.4 GetStreams

A `GetStream` is used to access large objects piece-by-piece, or in *spans*. Like a `PutStream`, it contains both a blocking and non-blocking form of the `GetSpan()` method.

Blocking Example:

```
// This example retrieves a large object from WOS and writes
//   it to a file piece by piece.
public void getBig(WosCluster wos, String oid, String filename) {
    try {
        OutputStream outstream = new BufferedOutputStream(
                                 new FileOutputStream(filename));
        WosGetStream gets = wos.createGetStream(oid);

        long length = gets.getLength();
        int chunksize = 32768;
        byte[] buffer = new byte[chunksize];
        long offset = 0;

        while (length > 0) {
            long take = length > chunksize ? chunksize : length;
            gets.getSpan(buffer, offset, take);
            outstream.write(buffer, 0, (int)take);
            length -= buffer.length;
            offset += buffer.length;
        }

        outstream.close();
    }
    catch (WosException e) {
        System.out.printf("Error reading object %s: %s\n", oid,
e.getMessage());
    }
    catch (IOException e) {
        System.out.printf("File writing object data to %s: %s\n", filename,
e.getMessage());
    }

}
```

Higher throughput may be achieved using the non-blocking form of `GetSpan` (since more work can be executed ahead, in parallel, in the cluster). Note, however, that non-blocking `GetSpan` callbacks may occur in any order, so some care may be necessary in some applications.

Non-Blocking Example:

```
public void getBigNB(WosCluster wos, String oid, String filename) {
    try {
        RandomAccessFile file = new RandomAccessFile("testout2.txt", "rw");
        WosGetStream gets = wos.createGetStream(oid);

        long length = gets.getLength();
        int chunksize = 32768;
        long offset = 0;

        while (length > 0) {
```

```
            long take = length > chunksize ? chunksize : length;
            gets.getSpan(offset, take, new GetSpanCB(file, offset,
(int)take));
            length -= take;
            offset += take;
        }
        wos.woswait();
        file.close();
    }
    catch (WosException e) {
        System.out.printf("Error reading object %s: %s\n", oid,
e.getMessage());
    }
    catch (IOException e) {
        System.out.printf("File writing object data to %s: %s\n", filename,
e.getMessage());
    }

}

private class GetSpanCB implements WosGetSpanCB {
    private RandomAccessFile file;
    private long offset;
    private int length;
    public GetSpanCB(RandomAccessFile file, long offset, int length) {
        this.file = file;
        this.offset = offset;
        this.length = length;
    }
    public void cb(WosStatus status, byte[] data) {
        if (!status.equals(WosStatus.OK)) {
            System.out.printf("Error getting span at offset %d: %s\n",
offset, status.getMessage());
            return;
        }
        try {
            file.seek(offset);
            file.write(data, 0, length);
        }
        catch (IOException e) {
            System.out.printf("Error writing data at offset %d: %s\n",
offset, e.getMessage());
        }
    }
}
```

2.6.5 GetStreams and Metadata

`GetStream`s support a number of metadata-specific features that are not available in the whole-object API's:

- It is possible to get an object without fetching its metadata
- It is possible to retrieve an object's metadata without fetching its data.

The `createGetStream()` method is overloaded to expose the prefetch_meta Boolean parameter, which is by default set to true, causing the metadata to be prefetched. To avoid prefetching the metadata, this parameter should be set to false. Even when this parameter is set to false, the metadata can still be explicitly retrieved with the `getMeta` call.

Example:

```
public void metaNoPrefetch(WosCluster wos, String oid) {
    try {
        boolean prefetch_meta = false;
        WosGetStream gets = wos.createGetStream(oid, prefetch_meta);
        // ...
    }
    catch (WosException e) {
        // ...
    }
}
```

Since creating a `GetStream` does not initiate retrieving object data (any of its spans) automatically, it is possible to access an object's metadata without retrieving data (by simply not calling `GetSpan`):

Example:

```
public void metaDataOnly(WosCluster wos, String oid) {
    try {
        WosGetStream gets = wos.createGetStream(oid);
        HashMap meta = gets.getMetaAll();
        Iterator itr = meta.entrySet().iterator();
        while (itr.hasNext()) {
            Map.Entry entry = (Map.Entry)itr.next();
            String key = (String)entry.getKey();
            String value = new String((byte[])entry.getValue());
            System.out.printf("Meta key=%s, value=%s\n", key, value);
        }
    }
    catch (WosException e) {
        // ...
    }
}
```

2.6.6 Stream Index Caching

Stream index caching can be used to maintain an internal cache of the index objects to avoid costly disk I/O for frequently accessed large objects.

Invoking `SetStreamCacheSize(int cache_size)` with the size argument greater than 0 (zero) will enable stream cache. Specify 0 (zero) disables the stream cache. By default, the stream cache is disabled.

The stream cache is finite in size and employs LRU (least recently used) policy for eviction. When the stream cache becomes full, the oldest item gets evicted from the cache in order to make way for the newest addition. Data can remain in caches after deletes until evicted.

2.7 Exceptions

All WOS exceptions derive from the WosException base class.

WOS Connect

- `CannotConnectException` -- Thrown if application cannot connect to the cluster

WOS Put/PutOID

- `InvalidPolicyException` -- A policy name was specified that is not defined in the cluster.
- `MaxMetaCountException` -- Each object supports a maximum of 32,000 pieces of metadata. A maximum of 64MB of metadata is supported.
- `ObjectFrozenException` -- Once a `WosObject` is `Put` into the cluster, it cannot be used to add another object into the cluster.
- `InvalidReservationException` -- An `OID` was specified that was not obtained via a call to `Reserve`, or that has already been written.
- `InvalidKeyNameException` -- Metadata keys must not start with ":"

WOS Get

- `ObjectNotFoundException` -- An `OID` was specified that refers to a non-existent object.

WOS Delete

- `ObjectNotFoundException` -- An `OID` was specified that refers to a non-existent object.

WOS Reserve

- `InvalidPolicyException` -- A policy name was specified that is not defined in the cluster.

WOS PutStream

- `StreamFrozenException` -- `PutSpan`s cannot be specified after a `PutStream` is closed. Metadata may not be set after a `PutStream` is closed. Further, a `PutStream` cannot be closed twice.
- `ObjectTooBigException` -- Attempted to write an object > 5 TB.
- `InvalidPolicyException` -- A policy name was specified that is not defined in the cluster.
- `MaxMetaCountException` -- Each object supports a maximum of 32,000 pieces of metadata which must "fit" within 64MB-512B of space.
- `InvalidReservationException` -- An `OID` was specified that was not obtained via a call to `Reserve`, or that has already been written.
- `InvalidKeyNameException` -- Metadata keys must not start with ":"

- `EmptyStreamException` -- Either a `PutSpan` attempted to write 0 bytes or the stream was closed with no data or meta data written

WOS GetStream

- `CannotAllocateException` -- A request for a span was made which will not fit in memory.
- `InvalidOffsetException` -- Offset specified is < 0 or > length of object.
- `ObjectNotFoundException` -- An `OID` was specified that refers to a non-existent object.

General

- `TooManyThreadsException` -- WOS cannot support more than two hundred user threads.

Chapter 3
Python API

3.1 Importing the WOS API

Stylistically, there are two basic choices for importing the API, depending on your preference for managing namespaces:

- Import all global classes into the current namespace

```
from wosapi import *

wos = WosCluster(...)
obj = WosObj()
...
```

- Import the `wosapi` module, while retaining it as a sub-namespace.

```
import wosapi

wos = wosapi.WosCluster(...)
obj = wosapi.WosObj()
...
```

Either style can be used and that choice does not affect the functionality of the library.

NOTE : GOA is only accessible via the RESTful HTTP API.

3.2 Installation

There are two WOS libraries required to run Python WOS scripts: `libwos_cpp.so` (the underlying C++ client library), and `wosapi.so` (the Python extension that provides access to the C++ library).

Both of these libraries may be installed at the system level (i.e. in `/usr/lib64/...`) or be installed locally to a user.

NOTE : WOS supports Python versions 2.6 and 2.7.

NOTE : RHEL6 64-bit libraries should be installed under `/usr/lib64`, whereas 32-bit libraries should be installed under (the parallel) `/usr/lib` paths. (Examples below assume 64-bit libraries).

If the libraries are installed locally, one must manually modify the Python module search path (using `sys.path.append(...)`) and the dynamic loader search path (using `export LD_LIBRARY_PATH=...`)

From the supplied `wos-*-py_sdk.tgz`, you may copy `libwos_cpp.so.1` to `/usr/lib64`. Once you do this, you should install a symbolic link between `libwos_cpp.so.1` and `libwos_cpp.so`:

```
# cp libwos_cpp.so.1 /usr/lib64
# cd /usr/lib64
# ln -s libwos_cpp.so.1 libwos_cpp.so
```

If you choose to leave it "local", arrange that `LD_LIBRARY_PATH` point to its full (absolute) path:

```
% export LD_LIBRARY_PATH=/home/user/fred/wos
```

From the supplied `wos-*-py_sdk.tgz`, you may copy `wosapi.so` to `/usr/lib64/python2.6/site-packages/wosapi.so`. If you choose to leave it "local", arrange to modify the Python extension search path to include the directory in which `wosapi.so` resides. For example, if you put your files in `/home/user/fred/wos`, then you may put the following before your `wosapi` import:

```
import sys
sys.path.append('/home/user/fred/wos')
...
from wosapi import *
...
```

3.3 Classes, Methods

WosCluster *cluster* [ctor]

The WosCluster constructor is used to connect to the cluster. The returned handle is used in all future Put, Get, and similar commands.

If all of the nodes in a cluster are configured under the same name in DNS, DNS will provide a round-robin response of different node IP addresses for successive connect statements, thereby achieving some measure of load-balancing.

```
wos = WosCluster("cluster-dnsname-or-ipaddr")
```

Only one connection to the cluster per process may be open at one time. Connect is a somewhat expensive operation. It is intended that applications do this once (or infrequently) and then keep their connection open while performing Put/Get/Delete operations on the cluster.

WosObj [ctor]

Create a WosObj

```
obj = WosObj()
obj.data = 'lots of bytes of data...'
obj.meta['a'] = 'b'
obj.meta['date'] = time.time()
```

put *object, policy* [method]

Store an object with replication directives specified by policy. The return value is an *object-id* (generally referred to as an OID).

```
policy = 'replicate'
oid = wos.put(obj, policy)
```

Get *oid* [method]

Retrieve an OID from the cluster

```
obj = wos.get(oid)

print obj.data
print obj.meta['date']
```

Delete *oid* [method]

Delete an object from the cluster. Note that although the space will be reclaimed for future use, the previously-used OID will never be returned again.

```
wos.delete(oid)
```

It is possible to break the Put step into two pieces which can be executed separately: the first step is to Reserve which returns an OID, and the second step is to use PutOID to associate an object with that OID. Note that for a given OID, PutOID can only be called once.

NOTE : If not all replicas of the object to be deleted are currently accessible to the client (for example, if there is a network failure to one zone), then the Delete operation will not be successful and a TemporarilyNotSupported error is returned.

Exists *oid* [method]

Check for existence of an object in the cluster. A successful response conveys a high level of confidence that the object data would be returned on `Get()`, as `Exists()` ensures that WOS is able to locate all portions of the referenced object. This call does not read the entire object off of disk, thus, there is a possibility that WOS will need to attempt correction of data integrity issues at the time the object is actually retrieved.

```
isExists = wox.exists(oid)
```

`isExists` will be `True` is if object exists otherwise it will be `False`.

Reserve *policy* [method]

Reserve an OID

```
policy = 'copy3'
oid = wos.reserve(policy)
```

PutOID *object, oid* [method]

Associate an object with a previously-reserved OID

```
wos.putoid(obj, oid)
```

NOTE : If not all replicas of the requested reservation are currently accessible to the client (for example, if there is a network failure to one zone), then the `PutOID` operation will not be successful and a TemporarilyNotSupported error is returned.

3.4 Streaming API

The streaming API allows reading and writing of objects which would not fit in memory all at once. New methods have been added to the `WosCluster` class which allow the creation of a `WosGetStream` and a `WosPutStream`.

`WosCluster.CreatePutStream` *`policy`* [factory method]

Note: Python is case-sensitive and the `CreatePutStream`, `CreatePutOIDStream`, and `CreateGetStream` methods have been defined in "camel-case" to match our other APIs more closely, and because these multi-word methods become hard to read if they are entirely lowercase.

Create a `WosPutStream` class. This method takes one argument—the policy to use for the `WosPutStream`. The `WosPutStream` and `WosGetStream` classes are intended to mimic the semantics of a file stream in Python as closely as possible.

`WosCluster.CreatePutOIDStream` *`oid`* [factory method]

The `CreatePutOIDStream` method takes one argument—namely a reserved `OID` that was obtained from a prior call to `WosCluster.reserve()`. It returns a `WosPutStream` - there is no `WosPutOIDStream` class, this method is simply a different factory for the `WosPutStream` class.

`WosPutStream.write` *`data`* [method]

The write method takes only one argument—the data to be written. This can be any Python binary-safe string. The data provided will be appended to any data that has already been written.

`WosPutStream.seek` *`offset`* [method]

The seek method takes one argument—the offset into the object that the next call to `write()` should start from.

`WosPutStream.meta` [method]

The meta member variable on the `WosPutStream` functions in the same manner as the meta member of the `WosObj`. It is a dictionary of string-based key/value pairs. Any data that has been added to this dictionary prior to calling `close()` will be saved with the object.

`WosPutStream.close()` [method]

The close method takes no arguments and returns the `OID` of the object that has been written to the cluster.

`WosCluster.CreateGetStream` *`oid, prefetch_meta`* [factory method]

Create a `WosGetStream` class. The `CreateGetStream` method takes one required argument—the `OID` of the object to be retrieved, and one optional boolean argument which indicates whether the metadata should be retrieved.

3.4.1 GetStreams and Metadata

GetStreams support a number of metadata-specific features that are not available in the whole-object API's:

- It is possible to get an object without fetching its metadata.
- It is possible to retrieve an object's metadata without fetching its data.

When opening a GetStream, there is an optional prefetch_meta Boolean parameter, which is by default set to true, causing the metadata to be prefetched. To avoid prefetching the metadata, this parameter should be set to false. Even when this parameter is set to false, the metadata can still be explicitly retrieved with the GetMeta call.

Since creating a GetStream does not initiate retrieving object data (any of its spans) automatically, it is possible to access an object's metadata without retrieving the data (by simply not calling GetSpan).

WosGetStream.read *length* [method]

The read method takes one argument—the number of bytes of data to be read. The WosGetStream has an internal concept of the "current offset" within the object's data. This is initialized to the beginning of the object (offset 0) when the WosGetStream is created, and can be modified with the seek method. The read method also increments the current offset by the amount read, such that repeated calls to read() will read the entire object. When the end of the object has been reached, read() will return an empty string.

WosGetStream.seek *offset* [method]

The seek method takes one argument—the offset into the object that the next call to read() should start from. If seek is called with an offset greater than the length of the object, an exception will be thrown.

WosGetStream.getlength() [method]

The getlength method takes no arguments and returns the length of the data portion of the object.

WosGetStream.getmeta *key* [method]

Unlike the WosObj returned by WosCluster.get(), where the entire metadata dictionary is immediately available, WosGetStreams allow you to avoid accessing the metadata, so an additional method is necessary if you wish to access individual metadata key/value pairs. The getmeta method takes one argument—a string containing the key for which the data should be retrieved. The data string is returned.

WosGetStream.getmetadict() [method]

Unlike the WosObj returned by WosCluster.get(), where the entire metadata dictionary is immediately available, WosGetStreams allow you to avoid accessing the metadata, so an additional method is necessary if you wish to access the dictionary containing all of the metadata stored on the object. The getmetadict method takes no arguments and returns this dictionary.

3.4.2 Multi-part Upload

Refer to Appendix A for description of the multi-part upload feature.

`WosCluster.CreateMultiPartPuttStream` *`policy`* [factory method]

The `CreateMultiPartPutStream` takes one argument, the policy to use. It returns a `WosPutStream`. Like `CreatePutOIDStream`, this factory method returns a `WosPutStream` instance.

`WosPutStream.writepart` *`oid`* [method]

This method is only allowed on `WosPutStream`s opened with `CreateMultiPartPutStream`. It appends the sub-object referenced by *`oid`* to the multi-part object immediately following the last part written.

3.4.3 Stream Index Caching

Stream index caching can be used to maintain an internal cache of the index objects to avoid costly disk I/O for frequently accessed large objects.

Invoking **`WosCluster.SetStreamCacheSize`** *`cache_size`* with the size argument greater than 0 (zero) will enable stream cache. Specifying 0 (zero) disables the stream cache. By default, the stream cache is disabled.

The stream cache is finite in size and employs LRU (least recently used) policy for eviction. When the stream cache becomes full, the oldest item gets evicted from the cache in order to make way for the newest addition. Data can remain in caches after deletes until evicted.

3.5 Examples

putfile.py:

```
#!/usr/bin/env python
import sys, optparse, time
from wosapi import *

def Put(wos, file, policy):
    MAXSIZE = 64*1024*1024-512

    f = open(file, 'rb')
    data = f.read(MAXSIZE) # for now, we deal only with smaller files
    f.close()

    # Make an object
    obj = WosObj()
    obj.data = data
    obj.meta['date'] = str(time.time())

    # Store it
    oid = ''
    try:
        oid = wos.put(obj, policy)
    except WosError, e:
        print e

    return oid

if __name__ == '__main__':
    # parse command-line args:
    usage = "usage: %prog -c cluster -p policy file"
    parser = optparse.OptionParser(usage)
    parser.add_option('-c', '--cluster', dest='cluster',
        default=None, help='WOS cluster dns or IP')
    parser.add_option('-p', '--policy', dest='policy',
        default='replicate', help='WOS Put policy')

    (options, args) = parser.parse_args()

    if options.cluster == None:
        print "Must specify cluster with -c"
        sys.exit(1)

    if len(args) != 1:
        print "Must specify a file..."
        sys.exit(1)

# Connect to WOS Cluster:
try:
    wos = WosCluster(options.cluster)
except WosError, e:
    print "Error during connect"
    print e
    sys.exit(1)

file = args[0]
oid = Put(wos, file, options.policy)
print "File %s stored to OID %s" % (file, oid)
```

Streaming Example 1:

putstreamfile.py:

```
#!/usr/bin/env python
import sys, optparse, time
from wosapi import *

def PutFileAsStream(wos, file, policy):
    CHUNKSIZE = 1024*1024

    f = open(file, 'rb')
    puts = wos.CreatePutStream(policy)
    puts.meta['filename'] = file
    puts.meta['date'] = str(time.time())

    while True:
        data = f.read(CHUNKSIZE)
        if len(data) == 0:
            break
        try:
            puts.write(data)
        except WosError, e:
            print "Error during WosPutStream.write()"
            print e
            sys.exit(1)

    f.close()
        try:
            oid = puts.close()
        except WosError, e:
            print "Error during WosPutStream.close()"
            print e
            sys.exit(1)

        return oid

if __name__ == '__main__':
    # parse command-line args:
    usage = "usage: %prog -c cluster -p policy file"
    parser = optparse.OptionParser(usage)
    parser.add_option('-c', '--cluster', dest='cluster',
        default=None, help='WOS cluster dns or IP')
    parser.add_option('-p', '--policy', dest='policy',
        default='replicate', help='WOS Put policy')

    (options, args) = parser.parse_args()

    if options.cluster == None:
        print "Must specify cluster with -c"
        sys.exit(1)

    if len(args) != 1:
        print "Must specify a file..."
        sys.exit(1)

# Connect to WOS Cluster:
try:
    wos = WosCluster(options.cluster)
except WosError, e:
    print "Error during connect"
    print e
    sys.exit(1)

file = args[0]
oid = PutFileAsStream(wos, file, options.policy)
print "File %s stored to OID %s" % (file, oid)
```

Streaming Example 2:

getstreamfile.py:

```
#!/usr/bin/env python
import sys, optparse, time
from wosapi import *

def GetStreamToFile(wos, file, oid):
    CHUNKSIZE = 1024*1024

    f = open(file, 'wb')
    gets = wos.CreateGetStream(oid)

    while True:
        try:
            data = gets.read(CHUNKSIZE)
        except WosError, e:
            print "Error during WosGetStream.read()"
            print e
            sys.exit(1)

        if len(data) == 0:
            break
        f.write(data)

    f.close()
    return

if __name__ == '__main__':
    # parse command-line args:
    usage = "usage: %prog -c cluster -o OID file"
    parser = optparse.OptionParser(usage)
    parser.add_option('-c', '--cluster', dest='cluster',
        default=None, help='WOS cluster dns or IP')
    parser.add_option('-o', '--oid', dest='oid',
        default='replicate', help='WOS Put policy')

    (options, args) = parser.parse_args()

    if options.cluster == None:
        print "Must specify cluster with -c"
        sys.exit(1)

    if options.oid == None:
        print "Must specify OID with -o"
        sys.exit(1)

    if len(args) != 1:
        print "Must specify a file..."
        sys.exit(1)

# Connect to WOS Cluster:
try:
    wos = WosCluster(options.cluster)
except WosError, e:
    print "Error during connect"
    print e
    sys.exit(1)

file = args[0]
GetStreamToFile(wos, file, options.oid)
print "OID %s written to file %s" % (options.oid, file)
```

Multi-part Upload Example:

```
import wosapi


def create_object_parts(wos, policy, parts):
    '''
    Given a list of strings, stores each string as a WOS object. Returns the
    corresponding OIDs in a list.
    '''
    oids = []
    for part in parts:
        obj = wosapi.WosObj(part)
        oid = wos.put(obj, policy)
        oids.append(oid)

    return oids


def create_multipart_object(wos, policy, oids):
    '''
    Given a list of OIDs, stores each OID in a multipart stream to assemble a
    larger object. Returns the multipart OID.
    '''
    puts = wos.CreateMultiPartPutStream(policy)

    for oid in oids:
        puts.writepart(oid)

    return puts.close()


def sample():
    '''
    This sample uses a multipart stream to create an object assembled
    from other parts.

    Sample output:

    Reading back object parts:
      Part 1: oid = 4D52hoizAoSITNCpspB3nXsECPbiO4xYtWduDteB, data = 'Hello'
      Part 2: oid = IAJGineDDYRHv9Ba_BX2iuOrNAnIVOClT6zqJOwK, data = ' '
      Part 3: oid = eAfQ6GbVDOJmqrBMxj2hiYrggBysWFZzE88oAGrO, data = 'World'
      Part 4: oid = oBpmjhGjC4QB3dAA9a_wsqz8kGpizSYz3hBdEvOM, data = '!'

    Reading back multipart object:
      Hello World!
    '''

    ip_addr = "wos7k12.sol"
    policy = "single"

    wos = wosapi.WosCluster(ip_addr)

    # store object parts
    parts = create_object_parts(wos, policy, ["Hello", " ", "World", "!"])

    print "Reading back object parts:"
    for i, part in enumerate(parts, start=1):
        print "  Part {0}: oid = {1}, data = '{2}'".format(
            i, part, wos.get(part).data)
    print ""

    # create multipart object from parts
    moid = create_multipart_object(wos, policy, parts)
```

```
    print "Reading back multipart object:"
    print " ", wos.get(moid).data


if __name__ == "__main__":
    sample()
```

Chapter 4
(HTTP) RESTful API

4.1 Introduction

The WOS REST interface is hosted on port 80 of every WOS node.

For WOS as an appliance, an alternate secure interface (based on HTTPS) is also available on port 443 of every WOS node. Note that I/O performance through HTTPS can significantly differ from HTTP. SSL encryption is performed on both client and server sides and depends on computational resources available on both. Thus performance can vary depending on your client hardware, ratio of small versus large objects being written or read, client distance to the cluster, and session length, for example.

Both port 80 and port 443 can be disabled by the cluster administrator.

It is generally intended that this protocol acts as an extension of a subset of RFC 2616 which describes HTTP/1.1. By this we mean that the behaviors (particularly as they apply to headers and caching, etc.) that are described in RFC 2616 are not meant to be implemented differently unless explicitly stated. We believe that RFC 2616 allows for a small-footprint subset, by implementing the RFC's "MUST rules" that intersect with the functionality we provide. So, for instance, if RFC 2616 says that a request MUST contain a Host header, we intend to comply with that. Notwithstanding this, many features of a full HTTP/1.1 server are not covered by our server.

We are careful to distinguish between HTTP PUT, GET, DELETE, and EXISTS messages and WOS PutObject, GetObject, DeleteObject, and ExistsObject operations.

Note that there are two conventions for performing GetObject operations (see below); one provides the OID as part of the URL; the other specifies it in a custom header.

The following features are incorporated:

- Additional HTTP Headers have been introduced to communicate return-values and status associated with the basic operations.
- Any node in the cluster can service HTTP requests. Requests are automatically directed to the node(s) that are best able to service the request without use of client redirects.
- Up to 1000 connections per node can be established concurrently

Helpful Hint:

- Best practice is for clients to evenly distribute requests to local nodes. This can be facilitated by creating a DNS entry for all nodes in the cluster or for all nodes in a given zone and using that in applications.
- Nodes send "keep-alive" packets to persist socket connections. For performance-sensitive applications, it may be best to keep a pool of established connections to nodes in the cluster.
- When more than 1000 concurrent REST connections are established with a node, the connection that makes the next HTTP request will receive a response "503 Service Unavailable" and its connection will be closed.

4.2 WOS Header Extensions

The following HTTP headers are specific to the DDN WOS HTTP Interface: (Header keywords, per RFC 2616, are case-insensitive).

x-ddn-status *value name* [header]

This header returns the WOS status associated with the previously executed command; value is a number which represents the error, whereas description is a textual version of the error. Note that these error values should not be confused with HTTP status; these values refer to the success of the underlying WOS operation. The values are consistent with the values returned by the C++ API.

Value	Name	Meaning
0	ok	Success
200	NoNodeForPolicy	No nodes will accept Put or Reserve operations for this policy
201	NoNodeForObject	No nodes have a copy of the requested object
202	UnknownPolicyName	Policy name or id is not currently supported by the cluster
203	InternalError	Unknown internal Error
205	InvalidObjId	An invalid OID was specified
206	NoSpace	The cluster is full
207	ObjNotFound	Object cannot be located
208	ObjCorrupted	Object does not match its checksum
209	FsCorrupted	Filesystem internal structures are corrupted
210	PolicyNotSupported	Insufficient cluster resources to service Put or Reserve request for this policy
211	IOErr	Unrecoverable drive error
212	InvalidObjectSize	> 5 TB
214	TemporarilyNotSupported	Operation should be retried momentarily
216	ReservationNotFound	Reservation not found for specified OID
217	EmptyObject	Attempt to store a zero-length object
218	InvalidMetadataKey	Invalid metadata key specified
219	UnusedReservation	Attempted Get of an unused reservation
220	WireCorruption	Uncorrectable network corruption of the object
221	CommandTimeout	Command did not complete in a timely manner
222	InvalidGetSpan	Illegal combination of buffered=true, integity=false
228	ObjectComplianceReject	Attempt to delete prior to immutable specified date
229	InvalidComplianceDate	An invalid compliance date was specified

x-ddn-oid *oid* [header]

This header specifies the WOS `OID` associated with the object recently reserved or put. This header is specified when performing `DeleteObject`, `PutOID`, `GetObject` operations. In these cases, the server returns this header as part of the response to provide context to the caller. Additionally, when performing a `PutObject` or `ReserveObject`, this header is returned in the response message to indicate the `OID` associated with the newly created object.

x-ddn-policy *policy* [header]

This header specifies the WOS policy used during `PutObject` and `ReserveObject` operations. Either the policy name (enclosed in quotation marks) or the policy ID may be used here.

x-ddn-meta [header]

This header specifies a list of metadata key=value pairs. Both keys and values MUST be quoted; to include a double-quote in a value, it must be preceded by a backslash. Multiple key-value pairs are separated by commas.

```
x-ddn-meta: "key1":"xyzzy", "key2 b":"something longer", "z":"the best"
```

- Keys must be < 64 characters in length.
- Keys and values must be text-only for REST metadata.
- Keys must not start with "`:`".
  NOTE: All keys beginning with "`:`" are reserved by WOS.

x-ddn-no-meta [header]

This header is optionally used when issuing `GetObject` commands; it specifies that metadata should not be returned with the object. Values for this header are true or false. Metadata is returned with all `GetObject` requests that do not specify otherwise via this header.

```
x-ddn-no-meta: true
```

x-ddn-put-on-close [header]

This header is optionally specified when issuing `PutOID` commands; it indicates that the data should be stored to the cluster if the connection is terminated, even if the total number of data bytes received thus far is less than the number specified in the Content-Length: header in the request. Values for this header are true or false. The default is false if the header is not specified.

x-ddn-put-idle-timeout [header]

This header is optionally specified when issuing `PutOID` commands; it indicates that the connection should be automatically terminated after the specified number of seconds has passed without receiving any data. Any time data is received, the timer is restarted.

The minimum value allowed is 5 seconds, and the maximum is 3600 seconds, or one hour. Specifying a negative value is illegal and will deactivate the feature, although an error will not be returned and the `PutOID` command will still proceed. Specifying a value greater than 3600 is also illegal and will deactivate the feature without triggering an error.

The optimal value for this timeout is the lowest value that you can tolerate without accidentally terminating connections that are still active. This threshold will be specific to your application/hardware/network/etc. and should be determined experimentally. Best practices would indicate adding a margin of safety to the determined value.

x-ddn-length *length* [header]

This header returns the length of the object stored with the specified WOS `OID` when performing the `GetObject` operation.

x-ddn-buffered [header]

This header is optionally specified when issuing `GetOID` commands. Internal to WOS, objects greater than 1 MB are broken up into 1 MB pieces. By default, when you read back a span of data, all 1 MB pieces that overlap the span are read into the memory space of the client and are cached there for a short while. It is often the case the next span requested is subsequent to the previous one, and so it will use the remainder of the previously cached 1 MB piece of the object. When set to true, which is the default setting, this header allows an efficient readback of a large object, span by span. Set to false to disable buffering, which is optimized for random access of small parts of large objects.

x-ddn-integrity-check [header]

This header is optionally specified when issuing `GetOID` commands. This is a performance optimization when reading back small spans of larger objects, because the integrity check operation requires the entire 1 MB piece to be read from disk and verified against its checksum. Disabling the integrity check allows WOS to read back only the data requested by the client. Note, however, that disabling the integrity check not only prevents WOS from identifying the requested data as corrupted, but it also bypasses the self-healing function. If you disable this integrity check, it is important that you have a way of identifying data corruption in your own application. Values for this header are true or false. The default is false if the header is not specified.

NOTE : The combination of "x-ddn-buffered: true" and "x-ddn-integrity-check: false" is not allowed.

---

x-ddn-retry-delete [header]

This header is optionally specified when issuing `DeleteObject` commands; it triggers WOS to retry failed deletes in the background. Values for this header are true or false. The default is false if the header is not specified.

x-ddn-is-multipart [header]

This header is optionally specified when issuing `PutObject` commands. Refer to Appendix A "Multi-part Upload" for feature description. Values for this header are true or false. The default is false if the header is not specified.

```
x-ddn-is-multipart: true
```

x-ddn-allow-redirect [header]

When this header is set on a `GET` request, it allows for an HTTP 302 Redirect response to be sent back to the client if more direct access to the target data is possible. This directive is only effective for data stored to single-instance ObjectAssure policies. If WOS determines that there is no benefit of a redirect, the `GET` request will be fulfilled as usual. Redirect is active when set to `true`.

4.3 WOS PutObject Operation

The `PutObject` command is used to store an object (and optionally associated metadata) to the WOS cluster. An `OID` is returned from all successful operations.

```
Request:
POST /cmd/put HTTP/1.1
content-type: application/octet-stream
content-length: 1234
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
x-ddn-policy: "replicate"
\r\n\r\n
...

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
```

To verify that your cluster is operational, you can use **`CURL`** to issue simple commands through the API:

To put an object:

```
curl -v --data "testing 1,2,3" -H x-ddn-policy:"United States" http://<node
IP address>/cmd/put
```

This will result in output similar to the following:

```
* About to connect() to 10.11.0.51 port 80
*   Trying 10.11.0.51... connected
* Connected to 10.11.0.51 (10.11.0.51) port 80
> POST /cmd/put HTTP/1.1
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5
OpenSSL/0.9.8b zlib/1.2.3 libidn/0.6.5
> Host: 10.11.0.51
> Accept: */*
> x-ddn-policy:United States
> Content-Length: 13
> Content-Type: application/x-www-form-urlencoded
>
> testing 1,2,3HTTP/1.1 200 OK
< x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
< Content-Type: application/octet-stream
< x-ddn-status: 0 ok
< Date: Thu, 30 Sep 2010 18:10:33 GMT
< Server: WOS/0.1
< Content-Length: 0
* Connection #0 to host 10.11.0.51 left intact
* Closing connection #0
```

Note that REST returned `HTTP code 200, x-ddn-status = "0 ok", and a value for x-ddn-oid.`

Notes:

- Partial Puts (with the Range: header) are not supported.

4.4 WOS GetObject Operation

The `GetObject` command is used to retrieve an object (and its associated metadata) from the WOS cluster by means of its `OID`.

```
Request: (OID specified in URL):
GET /objects/KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 1234
Content-Type: application/octet-stream
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
\r\n\r\n
...
```

OR:

```
Request: (OID specified in headers):
GET /cmd/get
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 1234
Content-Type: application/octet-stream
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
\r\n\r\n
...
```

Additionally, the `x-ddn-no-meta` header can be used to indicate that metadata should not be returned in the `GET` response. This may be useful if metadata is extremely large. The default action is to always return metadata; thus, if the `x-ddn-no-meta` header is *not* specified, or has a value other than true, metadata *will* be returned.

```
Request: (OID specified in headers):
GET /cmd/get
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
x-ddn-no-meta: true

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 1234
Content-Type: application/octet-stream
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
\r\n\r\n
...
```

To verify that your cluster is operational, you can use **CURL** to issue simple commands through the API:

To get the object (substitute the OID you received above for the OID used here):

```
curl -v http://10.11.0.51/objects/KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
```

This will result in the output similar to the following:

```
* About to connect() to 10.11.0.51 port 80
*   Trying 10.11.0.51... connected
* Connected to 10.11.0.51 (10.11.0.51) port 80
> GET /objects/KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD HTTP/1.1
> User-Agent: curl/7.15.5 (x86_64-redhat-linux-gnu) libcurl/7.15.5
OpenSSL/0.9.8b zlib/1.2.3 libidn/0.6.5
> Host: 10.11.0.51
> Accept: */*
>
< HTTP/1.1 200 OK
< x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
< Content-Type: application/octet-stream
< x-ddn-status: 0 ok
< Date: Thu, 30 Sep 2010 18:12:36 GMT
< Server: WOS/0.1
< Content-Length: 13
Connection #0 to host 10.11.0.51 left intact
* Closing connection #0
testing 1,2,3 $
```

Note that REST returned `HTTP code 200, x-ddn-status = "0 ok", and the contents of the object "testing 1,2,3"`.

4.5 Retrieving Metadata Only

To retrieve metadata for an object without reading back the contents of the object itself, use the `HEAD` request instead of `GET`.

```
Request: (OID specified in URL):
HEAD /objects/KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 1234
Content-Type: application/octet-stream
x-ddn-status: 0 ok
x-ddn-length: 1234
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
```

OR:

```
Request: (OID specified in headers):
HEAD /cmd/get
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 1234
Content-Type: application/octet-stream
x-ddn-status: 0 ok
x-ddn-length: 1234
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
```

4.6 WOS GetObject with Range

`GetObject` is the ONLY command which may optionally make use of byte-ranges. In such a case, the request header contains a `Range` header, while the response contains a `Content-Range` header.

Both `/cmd/get` and `/objects/$OID` URL's are supported.

```
Request:
GET /objects/KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
range: bytes=1000-1999

Response:
HTTP/1.1 206 Partial content
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 1000
Content-Range: bytes 1000-1999/1000000
Content-Type: application/octet-stream
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
\r\n\r\n
...
```

Notes:

- A single request with multiple ranges (implying a response with multipart/byteranges) is not supported.

4.7 WOS DeleteObject Operation

The `DeleteObject` command is used to remove an object from the cluster. Although future `PutObject` operations may reclaim the space occupied by the deleted object, its `OID` will never get re-used.

```
Request:
POST /cmd/delete HTTP/1.1
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
```

Notes:

- If not all replicas of the object to be deleted are currently accessible to the client (for example, if there is a network failure to one zone), then the `Delete` operation will not be successful and the `x-ddn-status` header value in the HTTP response will be set to TemporarilyNotSupported. If a `Delete` fails, the object should be considered to be in an indeterminate state. It may still be retrievable or it may be actually be completely deleted. The `Delete` should be re-issued by the application.
- If the header field `x-ddn-retry-delete: true` is specified, WOS will retry the failed deletes in the background and the HTTP status code of "202 AcceptedForProcessing" is returned.
- In the case of a deletion retry, the `x-ddn-status` header value in the response will indicate "0 ok" for successful requests.

4.8 WOS ExistsObject Operation

The `ExistsObject` command is used to check if an object exists in the cluster. A successful response conveys a high level of confidence that the object data would be returned on `Get()`, as `Exists()` ensures that WOS is able to locate all portions of the referenced object. This call does not read the entire object off of disk, thus, there is a possibility that WOS will need to attempt correction of data integrity issues at the time the object is actually retrieved.

```
Request:
GET /cmd/exists HTTP/1.1
Host:
x-ddn-oid: pAo0VBxmDOBLLxA1cf_HmZJPHCI28sPVWFLIJp_C

Response:
HTTP/1.1 204 No Content
x-ddn-oid: pAo0VBxmDOBLLxA1cf_HmZJPHCI28sPVWFLIJp_C
x-ddn-status: 0 ok
Date: Thu, 12 Mar 2015 02:24:26 GMT
Server: WOS/0.1
Content-Length: 0
```

4.9 WOS ReserveObject Operation

The `ReserveObject` command is used to reserve an `OID`, without associating data with the object. This `OID` can later be used **one-time-only** with the `PutOID` command.

```
Request:
POST /cmd/reserve HTTP/1.1
content-type: application/octet-stream
content-length: 0
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-policy: "replicate"

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
```

4.10 WOS PutOID Operation

The `PutOID` command is used to store data to a known `OID`, previously obtained by use of the `ReserveObject` command. For a given `OID`, this command may be executed only once.

```
Request:
POST /cmd/putoid HTTP/1.1
content-type: application/octet-stream
content-length: 1234
date: Tue, 01 Jan 2013 01:01:01 GMT
host:
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
\r\n\r\n
...

Response:
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2013 01:01:02 GMT
Server: WOS-1.1
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
```

Notes:

- If not all replicas of the requested reservation are currently accessible to the client (for example, if there is a network failure to one zone), the `PutOID` operation will not be successful and the `x-ddn-status` header value in the HTTP response will be set to TemporarilyNotSupported. If a `PutOID` fails, the `PutOID` should be re-issued by the application.

4.10.1 Interrupted PutOID

Normally, if the connection to the WOS HTTP server is terminated prior to sending the expected number of bytes of data, i.e. the amount specified in the Content-Length: header, the data will be discarded. This is the only mode of operation available for normal `PutObject` commands, since the `OID` is only returned upon completion of the command. For `PutOID`, however, the `OID` is already known, so another mode of operation becomes available. If the `x-ddn-put-on-close:` header is set to `true`, as described above, and the connection is closed, the object will be stored with the data sent to the HTTP server thus far, ignoring the Content-Length that was originally indicated.

Notes:

- This can be useful both in scenarios where the client machine or process crashes and the data sent thus far is still useful, and in cases where the client simply is not aware of the total length of the object at the beginning of the request. By specifying the `x-ddn-put-on-close: true` header, and an upper bound for the Content-Length: header, the HTTP interface allows a mode of operation more similar to the other language APIs where the final object size is not explicitly indicated. The only cost

associated with using the WOS HTTP server in this manner is that connections cannot be reused for subsequent requests, but this is likely not significant when storing large objects.

Depending on the client machine communicating with the WOS HTTP server, and the cause of the connection being terminated, TCP/IP may not indicate to the WOS HTTP server that the connection has been terminated until the socket times out. The TCP/IP socket idle timeout can be as long as one hour, so there can be a considerable delay before the data is made accessible via the `OID`. This period is also a period of risk because if the WOS HTTP server were to lose power before the data were fully committed, the data would be lost.

Using both the `x-ddn-put-on-close:` header and the `x-ddn-put-idle-timeout:` header, it becomes possible to not only store the data sent from the client, but also complete the transaction and make the data available via the `OID` within a reasonable (user-specified) period of time.

4.11 Multi-part Upload

Refer to Appendix A for description of the multi-part upload feature.

To use this feature, include the header directive `x-ddn-is-multipart:true` on a normal `PUT (/cmd/put)`. When this is specified, it is expected to find a JSON formatted body that enumerates the part OIDs as follows:

```
{
     "parts":[
        "DDCDHVz_DrI8JyC0iRexhK6NiMQq9nFgtEvHAd-D",
        "9C89-ysBCVxbWMDkMEdcG_f5jLM3KET79Zc_EtVM",
        "pAops7JVABjSzYB2df8Guhj05CbFRVRFqdgHCAfI",
        "cDdc7k0gD00NOtCeug-gd15MBNxQuDrYeP1PCYgA"
     ]
}
```

- The parts are concatenated in-order specified.
- There is a new HTTP response of “231 JsonParsingError” if the JSON in this call is malformed.
- If any of the part OIDs are invalid, the expected DDN result code is “205 InvalidObjId”.
- If the request body is empty, the DDN result code is “217 EmptyObject”.

Below is a full request and response example:

```
Request:
POST /cmd/put HTTP/1.1
content-type: application/octet-stream
content-length: 221
date: Tue, 06 Apr 2014 01:01:01 GMT
host:
x-ddn-meta: "a":"a b c", "b":"c d e", "contenttype":"image/jpeg"
x-ddn-policy: "replicate"
x-ddn-is-multipart: true
\r\n\r\n
{
   "parts":[
      "DDCDHVz_DrI8JyC0iRexhK6NiMQq9nFgtEvHAd-D",
      "9C89-ysBCVxbWMDkMEdcG_f5jLM3KET79Zc_EtVM",
      "pAops7JVABjSzYB2df8Guhj05CbFRVRFqdgHCAfI",
      "cDdc7k0gD00NOtCeug-gd15MBNxQuDrYeP1PCYgA"
   ]
}

Response:
HTTP/1.1 200 OK
Date: Tue, 06 Apr 2014 01:01:02 GMT
Server: WOS-1.1
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
x-ddn-is-multipart: true
x-ddn-status: 0 ok
x-ddn-oid: KBLf1Ys8A9zPJtB-R8xGd85-8EMGk_qotdV5JrCD
```

4.12 Stream Index Caching

Without stream index caching, wosrest exhibits stateless behavior. Every **`GET`** request is handled separately, and the corresponding objects or spans of an object are fetched from the disks. If a user is repeatedly fetching the same object or accessing different spans of the same object via wosrest, the stateless behavior is expensive and sub optimal.

To address this use case, a stream cache is embedded inside `woslib`. Any application linked with `woslib` can access the stream cache. The goal is to maintain an internal cache of the index objects to avoid costly disk I/O for frequently accessed large objects.

Implementation Details

- Each woslib instance maintains its private copy of the stream cache. Multiple woslib instances do NOT share or synchronize their respective stream caches.
- The stream cache is finite in size and employs LRU (least recently used) policy for eviction. When the stream cache becomes full, the oldest item gets evicted from the cache in order to make way for the newest addition.
- When enabled, `CreateGetStream` is the only operation that would add an object into the stream cache.
- WOS cluster operations like `CreatePutStream`, `CreateUpdateStream`, `Delete`, disk failure, and data rebuild do NOT have any effect on the stream cache.
- Life span of the stream cache is restricted to the life span of the application using the woslib. The stream cache does not persist across application restarts.
- By default, the stream cache is disabled.
- There is **no** corresponding `GetStreamCacheSize()` API.

Please contact DDN Support if you want to enable this feature.

Appendix A

Multi-part Upload

The following provides a description of the multi-part upload feature.

Multi-part Upload

Multi-part upload allows portions of an object to be written to WOS independently and then consolidated into a single accessible WOS object. The final consolidated object is referenced by a single OID for the purpose of retrieval or deletion, while the component sub-objects still remain accessible independently. The final multi-part object should be viewed as a holding references to the sub-objects, though there is no reference counting. This means that a deletion of a sub-object will render that portion of the multi-part object inaccessible. Similarly, deletion of the multi-part object will also delete all its sub-objects, rendering them inaccessible.

Any metadata placed on the part objects is not directly accessible via the consolidated multi-part object. However, metadata may be added to the multi-part object as is possible with any WOS object.

Multi-part upload is available for C++, Java, Python, and REST API types. The part objects may be any size or even other multi-part objects. Though the syntax varies across API types, the general flow is as follows:

```
CreateMultiPartPutStream(policy)
PutPart (oid1)
PutPart(oid2)
…
PutPart(oidN)
Close
```

Parts are appended in the order specified.

Notes

- Multi-part objects do not allow for gaps.
- Part objects must be accessible at the time the multi-part object is created, or the multi-part creation will fail.
- Multi-part upload is primarily targeted for use-cases where the parts themselves are large (>> 1 MB). Use of `PutPart` for parts less than 1 MB should be reserved for the last portion of a multi-part object, where it may be unavoidable.
- The aggregate multi-part object may not exceed the overall maximum WOS object size of 5 TB.

Contacting Technical Support

Please contact DDN Support if you have questions or require assistance. Support can be reached at any time by phone, by email, or on the web.

Web

Support Portal	https://community.ddn.com/login
Portal Assistance	webportal.support@ddn.com

Telephone

DDN Worldwide Directory	http://www.ddn.com/support/contact-support

Email

Support Email	support@ddn.com

Bulletins

Support Bulletins	http://www.ddn.com/support/technical-support-bulletins
End-of-Life Notices	http://www.ddn.com/support/end-of-life-notices
Bulletin Subscription Request	support-tsb@ddn.com

Product Shipping Instructions

If you are shipping the product to another location, always use the original packaging provided with your unit(s).

If you are sending a product to DataDirect Networks for warranty or out of warranty repair, you *must* obtain a Return of Materials Authorization (RMA) number from DDN Support.

Corporate Headquarters 9351 Deering Avenue, Chatsworth, CA 91311 | ddn.com | Phone: +1.818.700.7600 Fax: +1.818.700.7601