XRootD

Scalable Service Interface

3 June 2016

Andrew Hanushevsky

SLAC
NATIONAL ACCELERATOR LABORATORY

1 Introduction

This document describes the **S**calable **S**ervice **I**nterface (**SSI**) component of the **XRootD** framework. The **SSI** is a multi-threaded **XRootD** plug-in that implements a request-response framework. Using this framework you can write client applications that issue arbitrary requests to a server that executes the request and then responds with a result. Since the framework is based on **XRootD**, all of the base features available to any **XRootD** plug-in are available to be used by **SSI**. This includes server clustering which allows you to cluster an unlimited number of servers to achieve horizontal scaling.

Additionally, all of the actions taken by the framework that may involve a delay are asynchronous using a callback mechanism for serialization. That is, when an action is taken that may involve a delay, it is launched in the background and the requestor is told that the results of the action will be made available when a requestor-provided callback is invoked. This minimizes the number of threads needed to implement a service as the thread requesting an action is free to do other work while the action is executed. This maximizes scaling within a process.

The **SSI** framework implements a remote object execution model. In this model, actions taken by the client, such as object creation, are symmetrically performed at the server. In this model client actions appear, to the extent possible, to occur locally while the actual execution is remote. This simplifies writing complex client-server interactions and normally very little framework code is needed to implement a request-response service.

While **SSI** is implemented using **XRootD** protocol, the **SSI** related classes do not expose any **XRootD** dependencies. Thus, it is possible to port the framework to most other protocols without impacting existing service applications.

1.1 Key Concepts

The **SSI** is an object based framework using the following key concepts:

- one or more services,
- one or more resources,
- a session for each resource,
- one or more requests, and
- request responders.

A service is responsible for provisioning a resource. When a service is created, the initial point of contact (i.e. host and port) must be specified. The initial point of a contact can be a single node or the head node of a node cluster. Hence, a service is tied to one or more service providers. There can be any number of service providers, each with its own unique set of resources. Usually, there is only one service.

A resource is an abstract name of some capability offered by a service provider. It is up to the implementation to assign names to particular resources available via a service provider. The resource simply identifies what future requests will be using along with possible restrictions. The **SSI** framework uses the initial point of contact to locate a real server capable of handling the named resource and creates a session binding the resource name to a server providing that named resource. There can be any number of resource and sessions. However, there is always a 1-to-1 correspondence between a resource and a session.

Once a session is established, one or more requests using the resource can be executed within the context of the session. Once no more requests need to be issued, the resource and its corresponding session are un-provisioned by the client to free up all memory allocated to handling the interaction.

The request responders are a server-side concept needed to send a response in the context of a specific request. A response can be:

- fixed amount of arbitrary data,
- an arbitrary amount of streamed data,
- a file, or
- an error message.

The server side service implementation determines the appropriate response.

In its simplest form, resources can be provided by a single server. That server is responsible for processing requests and providing responses. The next few sections

assume such a setup as it is simpler to explain the object interactions using a single server. A more complex scheme is a setup of several servers with a mixture of resources, some replicated and some not, configured in a cluster. The concepts presented in the simple case apply to clustered servers with some additional classes to manage the cluster. This is covered later.

1.2 Key Classes

The following table outlines the classes used on the client as well as the server. In most cases these are the same. It is best to read the detail header file comments on how to use the classes. Examples shown in this document do not necessarily show all possible options.

Class	Header	Client	Server	Comments
XrdSsiCluster	XrdSsiCluster.hh	n/a	opt	Effects cluster actions
XrdSsiEntity	XrdSsiEntity.hh	n/a	opt	Describes an incoming client
XrdSsiErrInfo	XrdSsiErrInfo.hh	Y	Y	Describes any error
XrdSsiLogger	XrdSsiLogger.hh	n/a	opt	Route log messages
XrdSsiProvider	XrdSsiProvider.hh	Y	Y	Describes a service provider
XrdSsiRequest	XrdSsiRequest.hh	Y	Y	Embodies a client request
XrdSsiResource	XrdSsiResource.hh	Y	Y	Describes a resource
XrdSsiRespInfo	XrdSsiRespInfo.hh	Y	opt	Actual response
XrdSsiResponder	XrdSsiResponder.hh	n/a	Y	Responds to a request
XrdSsiService	XrdSsiService.hh	Y	Y	Describes a service
XrdSsiService:: Resource	XrdSsiService.hh	Y	Y	Use to provision a resource
XrdSsiSession	XrdSsiSession.hh	Y	Y	Describes a session
XrdSsiStream	XrdSsiStream.hh	opt	Y	Streams response data

In the table above, “n/a” indicates the class is not applicable in the given context and “opt” indicates that the class is optional in the given context (i.e. it may not be needed, depending on the service being provided).

Implementing a client that uses an existing service provider is the simplest way to start. The following sections provide a client-side example and show how actions on the client are reflected on a server. This provides a clear explanation on how to use these classes on a client and can be used to guide a remote service implementation.

This will be augmented in following sections that guide you through implementing a simple remote service.

1.3 Callbacks, Threads, and Mutexes

Programming in a callback-centric environment, while not difficult, can be challenging. This is because the callback is not only asynchronous, meaning it can happen at any time, there is no guarantee whether or not a separate thread is used to effect the callback. For instance, when a thread calls an **SSI** method that method may perform an immediate callback using the calling thread. If the caller is holding a mutex that must also be locked in the callback method, a deadlock will occur.

To avoid such scenarios do one of the following:

- do not hold any mutexes when calling any **SSI** method that responds with a callback, or
- use recursive mutexes.

Following this simple formula will avoid hours of debugging time.

The amount of parallelism is limited by the number of threads that can be active at any one time. This occurs at two levels: 1) threads used for callbacks, and 2) threads used to handle network traffic. By default, no more than 300 callbacks may be active and up to 30 threads are used to handle network traffic. Since network traffic is handled without blocking, a number equal to 10% of callback threads is usually sufficient. You can change the settings using the **XrdSsiProvider:: SetCBThreads()** method before making any **SSI** calls.

1.4 Timeouts

In many systems requests are bounded by timeouts (i.e. a request can only take so long before a retry is attempted or the request is deemed to have failed). Because the **SSI** framework allows almost any type of service implementation, there is no appropriate default timeout. Instead, all request related methods can specify a specific timeout as needed, Additionally, default timeouts can be set using the **XrdSsiProvider:: SetTimeout()** method before making any **SSI** calls. The **SSI** framework initially sets all timeouts to 24,855 days (essentially infinite).

2 The SSI Client

In a client application, the **SSI** framework already provides a service that allows the client to use remote service providers. This service merely coordinates the interactions between the client and the remote server and should be viewed as simply an extension of the remote service provider itself. In order to get access to client-side service coordinator the application must link with **libXrdSsi.so** that contains the framework implementation.

2.1 Step 1: Obtain a Service Provider

The first step is to obtain a service object that corresponds to a service provider. This is done using the built-in provider object. The following code fragment shows how to do this.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

extern XrdSsiProvider *XrdSsiProviderClient;
       XrdSsiErrInfo   eInfo;
       XrdSsiService  *theSP;
       char            contact[256];

strcpy(contact, "somehost:1234";

theSP = XrdSsiProviderClient->GetService(XrdSsiErrInfo eInfo,
                                         contact);

if (!theSP) {…} // eInfo has the reason for the failure
```

Here, contact is a pointer to a null terminated character array that holds the location of the service provider (e.g. *hostname:port*) such as "myprovider:1094"). This is the initial point of contact for future resource provision requests. The initial point of contact is not validated at this time. However, other errors may occur that cause the call to return a nil pointer. If a nil pointer is returned the **eInfo** object contains an error message along with an **errno** value describing why the call failed.

At this point the client is able to provision a resource.

2.2 Step 2: Derive a Resource Class

The next step is to define a resource. Resources are defined using the imbedded **Resource** class in **XrdSsiService**. The resource definition will be handed off to the service that was obtained in step 1 for provisioning. Since provisioning can take some time, the resource object requires that you define a callback that will be invoked when resource provisioning successfully completes or fails.

In the framework, callbacks are defined by inheriting the base class that you need to use and implementing the pure abstract callback method within that class. Looking at **XrdSsiService::Resource**, that method is called **ProvisionDone()** and may or may not called using a new thread. The following code fragment shows a sample implementation.

```
#include "XrdSsi/XrdSsiService.hh"

class myResource : public XrdSsiService::Resource
{
public:

virtual void ProvisionDone(XrdSsiSession *sessP);

             myResource() {}
virtual     ~myResource{} {}
};

myResource::ProvisionDone(XrdSsiSession *sessP)
{
   if (!sessP) {…} // Inherited eInfo has the failure reason
      else     {…} // We can now use the resource via sessP
}
```

The above code merely defines a resource class that can be used to instantiate a resource instance. So, it can be reused for resources that need to be provisioned. This is shown in the next step.

In the above example it is assumed that the resource will be used within the context of **ProvisionDone()** which may be inconvenient. There are many ways to avoid this but all require some kind of external serialization that is beyond the scope of this example.

2.3 Step 3: Provision a Resource

Now that there is a resource class with a callback, it can be used to instantiate a resource name and then provision it. In the code fragment below we use the results of the first two steps.

```
// theSP is set via step 1 (i.e. define a service provider)
//
XrdSsiService *theSP;

// myResource was defined in step 2.
//
myResource theResource("/mysql");

// Provision the resource with the name "/mysql". The result
// is returned via the resource object's callback method.
//
theSP->Provision(&theResource);

// This thread is free to do anything it wants now!
```

Note that provisioning a resource is an asynchronous operation. The **myResource::ProvisionDone()** method is invoked when the operation completes. Upon success it is given a pointer to the corresponding **XrdSsiSession** session object. This object is used to dispatch requests against the provisioned resource. The thread that invoked **Provision()** can be reused, returned to the thread pool, or terminated.

If your application is written to execute requests using single disposable sessions, you can use a short-cut **XrdSsiRequest** method and skip manually provisioning a resource. Refer to Single Session Requests on how to do this.

2.3.1 Resource Names

While resource names are arbitrary, by default they must start with a slash. It is possible to configure an **XRootD** server to accept names without a leading slash. The configuration option is covered under server resource names.

2.4 Step 4: Derive a Request Class

Much like the resource object, the request object must be defined as a derivation of the abstract **XrdSsiRequest** class. This is because it also has an abstract callback method (i.e. **ProcessResponse()** and optionally **ProcessResponseData()**) as well as an abstract method for the framework to obtain the request data (i.e. **GetRequest()** and optionally **RelRequestBuffer()**) to be sent to the server via the session object. The following minimal code snippet shows an example of how to do that.

```
#include "XrdSsi/XrdSsiRequest.hh"

class myRequest : public XrdSsiRequest
{
public:

virtual char *GetRequest(int &dlen)
                        {dlen = reqBLen; return reqBuff;}

virtual bool  ProcessResponse(const XrdSsiRespInfo &rInfo,
                                    bool            isOK);

virtual void  ProcessResponseData(char *buff, int blen,
                                  bool  last);

              myRequest(char *buff, int blen)
                       : reqBuff(buff), reqBLen(blen) {}
virtual      ~myRequest{} {}

private:
char         *reqBuff;
int           reqBLen;
};

bool myRequest::ProcessResponse(const XrdSsiRespInfo &rInfo,
                                      bool            isOK)
{
   if (!isOK) {…} // Inherited eInfo has the failure reason
      else    {…} // rInfo holds the response
   return true;
}

void myRequest::ProcessResponseData(char *buff, int blen,
                                    bool  last)
{ // Example implementation shown later }
```

Here we define the **GetRrequest()** method to simply return the pointer to the buffer holding the request data and the length of the request data. It is up to the implementation to create request data, save it in some manner, and provide it to the framework when **GetRequest()** is called. The optional **RelRequestBuffer()** method can be used to minimize memory usage as it is called once the framework no longer needs access to the request data (see the header file for details). How you generate the request data or supply it to the **GetRequest()** call is up to you. However, be aware that the thread used to initiate a request may be the same one used in the **GetRequest()** call and this may affect your implementation choice.

Since a request is asynchronously sent to a server via the session object, the **ProcessResponse()** callback method is used to inform the request object that the requested completed or failed. At that point you are assured that a response is actually ready. This method is called on a new thread.

The optional **ProcessReesponseData()** is another callback method that is used in conjunction with the request's **GetReesponseData()** method or when the response is a data stream and you wish to asynchronously receive data via the stream. Most, but not all, applications will need to implement a **ProcessReesponseData()** method since a) it is more convenient to use **ProcessReesponseData()**, and b) scalable applications generally require that any large amount of data be asynchronously received. Hence, the example shows an implementation of **ProcessReesponseData()**.

The next step involves issuing a request. The details of handling the response are covered afterwards.

2.5 Step 5: Initiate a Request

Requests are always executed in the context of a session. Recall that the session is tied to a particular resource which means that the request must be appropriate for that resource. The definition of appropriate is defined by the server-side implementation. So, your requests need to correspond to what the service provider allows. Violating that will likely return an error. Based on the preceding steps, the following code fragment shows how to initiate a request.

```
#include "XrdSsi/XrdSsiRequest.hh"

XrdSsiSession *sessP; // Set by a call to Provision()
XrdSsiRequest *reqP;  // Used for demonstration purposes

// Create a request object (upcast from your implementation)
//
reqP = new myRequest(reqData, reqDLen);

// Hand off the request to the session object
//
sessP->ProcessRequest(reqP);

// The thread is now free to do anything it wants!
```

Once you call **ProcessRequest()** you are effectively transferring control of the request object to the session object. This means

- you must not alter or delete the request data until after **RelRequestBuffer()** method is called, If you have not implemented this method, then you must not alter or delete the request data until the **ProcessResponse()** callback is invoked.
- You must not delete the request object before the request is finished (see finishing a request). This also applied to the request data if you did not implement a **RelRequestBuffer()** method.

Object ownership is used by the **SSI** framework to minimize data copying.

If your application is written to execute requests using single disposable sessions, you can use the **XrdSsiRequest:SSRun()** method to quickly initiate a request with minimum fuss. Refer to Single Session Requests on how to do this.

2.5.1 Multiple Simultaneous Requests

A session object is capable of handling multiple requests. Each request is sent to the server the session object is bound to. All you need to do is allocate a new request object and call the session's **ProcessRequest()** method. The requests are asynchronously managed and are executed in an arbitrary order. Currently, you can have up to 1024 outstanding requests per session object. Any attempt to exceed the limit will return an out-of-memory error.

It is more efficient to reuse a session object than to always allocate a new one for each request. However, that may complicate request management in your application. So, you should carefully judge the trade off. See the next section for additional information.

2.5.2 Single Session Requests

If your application requirements fit into a single session request framework (e.g. the cost of executing the request far outweighs the cost of provisioning and un-provisioning a resource) then you can use the **XrdSsiRequest::SSRun()** methods to drastically simplify request handling. You still need to specify a service and a resource but otherwise all the mechanics or provisioning, managing a session, and un-provisioning are handled for you. All responses are vectored through request's response callback methods (see the next section).

There are two variants of **XrdSsiRequest::SSRun()**. The first takes a pointer to a service object and pointer to a resource description defined by the **XrdSsiResource** class (see **XrdSsiResource.hh**). This gives you complete flexibility on how a resource is described. The second dispenses with a pointer to **XrdSsiResource** and, instead, takes a pointer to a resource name and an optional resource user. If you don't need more than a name and possibly a user then this is the simplest approach. In either case, when you call **XrdSsiRequest::SSRun()** it

- Provisions the resource via **XrdSsiService::Provision()**.
- Executes the request embodied by the request object using the returned session object.
- When the request completes (or any errors arise) the request's **ProcessRespnse() c**allback is invoked and you can do the appropriate response handling.
- When you are done you simply call request's **Finished()** method and the session is automatically un-provisioned.

2.6 Step 6: Process the response

After a server process the request it responds with the result. The response is always contained in the **XrdSsiRespInfo** object, a private member in the **XrdSsiRequest** object. That **XrdSsiRespInfo** object is passed to your implementation of **ProcessResponse()**. The following table lists all the possible response types you may receive relative to the object's member contents.

rType is	**buff**	**blen**	**eMsg**	**eNum**	**strmP**
isData	→ data buffer	buffer length	n/a	n/a	n/a
isError	n/a	n/a	→ message	errno	n/a
isStream	n/a	n/a	n/a	n/a	→ XrdSsiStream object

There are only three possible types of response that a client may see:

- a data buffer when **XrdSsiRespInfo::rType** is set to **isData**,
- an error when **XrdSsiRespInfo::rType** is set to **isError**, and
- a data stream when **XrdSsiRespInfo::rType** is set to **isStream**.

When the response is an error, the information in the **RespInfo** object is also copied to the **XrdSsiRequest**'s public **eInfo** member for consistency. Otherwise, the only other possible response is actual data that the server sent. That data is provided to the client either directly via a buffer or via a stream object. You may always obtain the response data using the request's **GetResponseData()** method. However, if efficiency is of concern, you should inspect the response type and act accordingly.

If the response type is **isData** then you should directly access the **RespInfo buff** and **blen** members to extract the response without copying data. If the response type is **isStream** then you can use **GetResponse Data()** or directly use the **XrdSsiStream** methods in the returned stream object via the **RespInfo strmP** member (either approach is equally efficient).

That said, it is easier to use the **GetResponseData()** method, as shown below. It allows you to avoid dealing with the details of the response type. Should you wish to directly use the stream object, see the description of this object either in its header file or refer to the section explaining streams.

```
#include "XrdSsi/XrdSsiRequest.hh"

bool myRequest::ProcessResponse(const XrdSsiRespInfo &rInfo,
                                      bool            isOK)
{static const int myBSize = 1024;

 if (isOK) {char *myBuff = malloc(myBSize);
            GetResponseData(myBuff, myBSize);
           }
    else {/* Handle error using rInfo.eNum & rInfo.eMsg */
          Finished(); // You can now do a "delete this"
         }
return true;
}
```

The previous code snippet shows a simple response handler. Notice that when the **isOK** argument is false then the response is an error (i.e. **rInfo.rType == isError**). Otherwise, the response is data via a buffer or a stream. In the example, we allocate a data buffer of some arbitrary size and call **GetResponseData()** passing it the buffer and its size. The response data will be placed in the buffer. How you actually handle response data buffer is up to you. In any case, you should always return true.

Once response data is placed in your buffer the request object's **ProcessResponseData()** method is called. You need to implement this method as part of your request object. A sample implementation is shown below.

```
#include "XrdSsi/XrdSsiRequest.hh"

XrdSsiRequest::PRD_Xeq myRequest::ProcessResponseData
                      (char *myBuff, int myBLen, bool isLast)
{
static const int myBSize = 1024;

// Process the response data of length myBLen placed in myBuff
// If there is more data then get it, else free the buffer and
// indicate to the framework that we are done
//
if (!isLast) GetResponseData(myBuff, myBSize);
   else {free(myBuff);
         Finished(); //You can now do a "delete this"
        }
return XrdSsiRequest::PRD_Normal;
}
```

The **ProcessResponseData()** method is passed the original buffer you supplied to **GetResponseData()** as well as the amount of data placed in the buffer. If you want to reuse this buffer you will need to track its size in a better way than shown. The **isLast** argument is set to true to indicate that there is no more response data. Otherwise, the supplied buffer was not large enough to contain all of the data and you can call **GetResponseData()** again to obtain it.

Be aware that the data response may zero-length. This can happen when the response only contains metadata or your service has no relevant response data. See the following section title "Metadata-Only Response".

The examples show that after the response is handled the request's **Finished()** method is called. This is a critical call and is explained in the section titled "Finish or Cancel the Request". Keep in mind that once **Finished()** is called, you must not reference any **RespInfo** object members as these are deleted as part of finishing the request.

Additionally, **ProcessResponseData()** must return one of the **PRD_Xeq** enums defined in **XrdSsiRequest**. Typically, **PRD_Normal** is always returned to indicate that normal post-processing is desired. It is possible to return an indication that the callback is to be held. This is covered under Pacing Responses.

2.6.1 Obtaining Optional Metadata Ahead of the Response

The **SSI** framework allows a server to send metadata ahead of the response data. This metadata may be used for any purpose. For instance, the metadata may describe the response so that it can be handled in the most optimum way.

You obtain any sent metadata with the **XrdSsiRequest::GetMetadata()** method using the request object associated with the response. Typically, you should get metadata in the **ProcessResponse** callback method before you issue **XrdSsiRequest::GetResponseData()**.

The metadata, if any, is persistent until you call **XrdSsiRequest::Finished()** method. After that call you must not reference the buffer holding the metadata.

2.6.2 Handling a Metadata-Only Response

The **SSI** framework allows a server to send only metadata as a response (i.e. the response only consists of metadata). This is useful for sending short responses and avoiding additional client-server handshakes to determine that there is no actual

response data other than the metadata. A metadata-only response is indicated when the response type is **isData** and the length of the data is zero.

There is no hard and fast rule in using metadata-only responses to avoid communication overhead. The **SSI** framework tries to avoid such overhead for relatively small data responses as well. See the section titled "Sending Only Metadata" for more information.

You obtain any sent metadata with the **XrdSsiRequest::GetMetadata()** method using the request object associated with the response. Typically, you should get metadata in the **ProcessResponse** callback method. You should *not* call **XrdSsiRequest::GetResponseData()** as there is no other response data to be received. A metadata-only response is indicated when **XrdSsiRespInfo::rType** is set to **isNil** (i.e. no response data is present).

The metadata, if any, is persistent until you call **XrdSsiRequest::Finished()** method. After that call you must not reference the buffer holding the metadata.

2.6.3 Pacing Responses

If you need to delay handling response data (e.g. it is too large relative to other things that need to occur), the **SSI** framework allows you to postpone the **ProcessResponseData()** callback to a later time. You do this by returning

XrdSsiRequest::PRD_Hold
: The callback is placed in a global hold queue and releases the thread for other work. Responses in the queue are restarted upon request in FIFO order.

XrdSsiRequest::PRD_HoldLcl
: The callback is placed in a local queue associated with the request identifier passed to the **XrdSsiRequest** constructor when the request object was allocated. If there is no request identifier, the callback is placed in the global queue. Responses in the queue are restarted upon request in FIFO order.

Calling the static method **XrdSsiRequest::RestartDataResponse()** restarts one or more **ProcessResponseData()** callbacks. The method accepts the number callbacks to restart and an optional request identifier. If a request identifier is not specified, the global queue is used. Otherwise, the queue associated with the specified request identifier is used.

When the **ProcessResponseData()** callback is restarted it is called with the same arguments before it was suspended.

2.7 Step 7: Finish or Cancel the Request

Recall that when you passed your request object to the session's **ProcessRequest()** method, ownership of the object transferred to the session object and you were not allowed to delete the request object. The request's **Finished()** method (supplied by the framework) is used to regain control of the object. It may be called at any time but it must be called and called only once. When you call **Finished()** before the response is fully processed (i.e. before your **ProcessResponse()** method is called or there is still outstanding response data) the request is considered to be cancelled. Good programming practice requires that you explicitly indicate cancellation by passing true as an argument to **Finished()**. When Finished is improperly called, it returns false to indicate that it is likely that your program has a logic error. Otherwise, it returns true.

```
#include "XrdSsi/XrdSsiRequest.hh"
•••
if (!Finished()) abort()
   else delete this;
•••
```

In the above code snipped, calling **Finished()** returns ownership of the request object back to the caller. At this point you can delete the request object. If your request objects are uniform you should consider reusing them to avoid repeated object allocations. A handy public member, **XrdSsiRequest::nextRequest** pointer, can be used to chain free request objects.

You should also be aware that when you use **Finished()** to cancel a request, the cancellation may not guaranteed to be reflected to the server. This occurs when, from the server's perspective, the request has indeed finished because all remaining response data is already in transit.

2.8 Step 8: Unprovision the Resource

When you fully done using a resource (i.e. there are no more requests using that provisioned resource) you should unprovision it. Since there is a 1-to-1 correspondence between a resource and a session object, the session's **Unprovision()** method is used to free up the resource as well as to delete the session object (you may not use delete on a session object). However, if there are still any outstanding requests (i.e. request objects whose **Finished()** method has not yet been called) the **Unprovision()** fails and returns false. This is a good indication of a logic error in your program.

2.9 Deleting Client-Side Objects

The following rules apply to safely delete the client-side objects described in the previous sections.

2.9.1 XrdSsiService Object

The **XrdSsiService** object cannot be explicitly deleted. To delete this object you must call its **Stop()** method which deletes the object if it is safe to do so. A service object can only be deleted after all resources provisioned using that object have been unprovisioned. Any attempt to stop the service while there are still provisioned resources causes the **Stop()** method to return false and the object is not deleted.

2.9.2 XrdSsiService::Resource Object

The resource object may only be deleted after its **ProvisionDone()** method is called. This means you can safely delete the object in the callback immediately before returning from the callback.

2.9.3 XrdSsiSession Object

The **XrdSsiSession** object cannot be explicitly deleted. To delete this object you must call its **Unprovision()** method which deletes the object if it is safe to do so. A session object can only be deleted after all requests handed to that object have completed (i.e. the **Finished()** method has been called for each one). Any attempt to unprovision the session while there are still unfinished requests causes the **Unprovision()** method to return false and the object is not deleted.

2.9.4 XrdSsiRequest Object

The request object can only be deleted after its **Finished()** method has been called. Unlike other objects, there is no safeguard from deleting the object prior to calling **Finished()**. Violating this rule is likely to cause an invalid memory reference.

3 The SSI Server

In a server, the **SSI** framework loads your service application as a plug-in. This means your code needs to be packaged as a dynamically loadable shared library.

Referencing the previous flow summary, it should become apparent that actions taken by the client are essentially recreated on the server. The following sections explain what the service application needs to provide.

3.1 Step 1: Define the Service Provider

Server-side services are provided via the **XrdSsiProvider** object. While the client-side has one that is built-in the server-side must define one. There are actually two types of providers:

- A **cmsd** provider that needs only to ascertain whether or not a resource is available on the node on which its **QueryResource()** method is called, and
- A **xrootd** provider that actually provides the service via the **GetService()** method. The provider still needs to be able to ascertain resource availability.

This section only describes the **xrootd** provider. You need not have a **cmsd** provider if you are not clustering your servers. Defining a resource lookup provider for a clustered environment is described in a subsequent section.

The service provider is used by **xrootd** to actually process client requests. The service provider object is pointed to by the global pointer **XrdSsiProviderServer** which you must define and set at library load time (i.e. it is a file level global static symbol).

When your library is loaded, the **XrdSsiProviderServer** symbol is located in the library. Initialization fails if the appropriate symbol cannot be found or it is a nil pointer.

The three methods that you must implement in your derived **XrdSsiProvider** object are:

- **GetService()**,
- **Init()**, and
- **QueryResource()**

The **QueryResource()** method is used to obtain the availability of a resource. This method may be called whenever the client asks for the resource status. The following code snippet shows how you would typically define the provider pointer at file level.

```
#include "XrdSsi/XrdSsiProvider.hh"

class MyServerProvider : public XrdSsiProvider {•••};

XrdSsiProvider *XrdSsiProviderServer = new MyServerProvider;
```

Once the provider object is found, its **Init()** method is called. The method should initialize the object for its intended use. Subsequently, a one-time call is made to its **GetService()** method to obtain an instance of an **XrdSsiService** object.

The following page shows a sample derivation of an **XrdSsiProvider** class.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

class myServerProvider : public XrdSsiProvider
{
public:

// The GetService() method must supply a service object
//
XrdSsiService *GetService(XrdSsiErrInfo &eInfo,
                          const char    *contact,
                          unsigned int   oHold=256
                         );

// Init() is always called before any other method
//
bool           Init(XrdSsiLogger *logP,  XrdSsiCluster *clsP,
                    const char   *cfgFn, const char    *parms,
                    int           argc,  char         **argv

                   ) {●●● initOK = true; // If all went well
                      return initOK;
                     }

// The QueryResource() method determines resource availability
//
XrdSsiProvider::
rStat          QueryResource(const char *rName,
                             const char *contact=0
                            );

               myServerProvider() : initOK(false);
virtual       ~myServerProvider() {}

private:

bool initOK;
};
```

3.1.1 XrdSsiProvider::Init() Arguments

The **Init()** method is called once after your shared library is loaded and before any other calls. The arguments that may be of interest are:

XrdSsiLogger **logP*

logP points to a generalized message routing object. You can use this object to include messages in the framework log file. Messages are automatically prefixed with a time stamp and, when relevant, the thread ID issuing he message. See **XrdSsiLogger.hh** include file for full details.

XrdSsiCluster **clsP*

clsP points to an object that is used to control server selection of and manage resources that the server may have. It is only relevant in a clustered environment. Cluster management relative to this object is described later.

const char **cfgFn*

cfgFn points to the name of the configuration file used to initialize the server. You can add your own directives to this configuration file and parse them out during initialization. This allows you to have a single configuration file.

const char **parms*

parms, if not nil, points inline parameters specified on the directive that identified your shared library. Passing parameters in this way is purely optional.

int *argc*, **char** ***argv*

The *argc* and *argv* parameters have the same use as the ones passed to **main()**. However, these command line arguments have been filtered out to only contain arguments specific to your plug-in. Refer to the section on starting **SSI** daemons for more information.

3.1.2 XrdSsiProvider::QueryResource() Arguments and Return Values

Whenever an **SSI** daemon needs to know the status of a resource it calls **QueryResource()**. This is true of the xrootd and, in a clustered environment, the cmsd. The argument, *rname*, passed is exactly the same as specified by the client when it created a derived instance of the **XrdSsiService::Resource** object in order to provision the named resource. The **QueryResource()** method should return one of three values:

XrdSsiProvider::notPresent
: The return value indicates that the resource does not exist.

XrdSsiProvider::isPresent
: The return value indicates that the resource exists.

XrdSsiProvider::isPending
: The return value indicates that the resource exists but is not immediately available. This is only useful in clustered environments where the resource may be immediately available on some other node.

The second argument, *contact*, is always nil for a server. It is only used by a client initiated query for a resource at a particular endpoint.

3.2 Step 2: Derive a Service Class

The provider object supplies a service object via the **GetService()** method. While the client has the option of obtaining multiple service objects, one for each service supplier end-point; that is meaningless for a particular server since there can only be one service relative to an end-point. Regardless, you must supply such a service object. Below is a sample derivation of a **XrdSsiService** class a derived **XrdSsiProvider** method; showing what each returns.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

class myService : public XrdSsiService
{
public:

void   Provision(Resource       *resP,
                 unsigned short  timeOut=0
                );
•••
};

XrdSsiService *myProvider::GetService(XrdSsiErrInfo &eInfo,
                                      const char    *contact,
                                      unsigned int   oHold=256
                                     )
               {if (initOK)return new myService(…);
                eInfo.Set("initialization failed!", EINVAL);
                return 0;
               }
```

This object returned by your provider is used as an **XrdSsiSession** object factory. Only one service object is ever obtained from the provider. The only method that your derived **XrdSsiService** object must implement is:

- **Provision()**

The service's **Provision()** method, upon success, must return a session object via the callback in the resource object. This is exactly the same sequence that the client follows. If the provision call fails then the callback is given a nil session object with its **eInfo** object containing the reason for the failure. The exact steps are described under resource provisioning.

3.3 Step 3: Derive a Session Class

Upon success, the **Provision()** method in your service object must provide a session object. The session object must inherit the **XrdSsiSession** class. Session objects are used to process client requests relative to the provisioned resource. If your session object will also be responding to a client request, it must also inherit the **XrdSsiResponder** class. Any object that has inherited the **XrdSsiResponder** class can post a response to a request. However, be aware that only one such object can bind its responder to a request at any one time. This mechanism is used to prevent inadvertent responses or responses to the wrong request. A sample derivation is shown

```
#include "XrdSsi/XrdSsiSession.hh"
#include "XrdSsi/XrdSsiResponder.hh"

class mySession : public XrdSsiSesesson,
                  public XrdSsiResponder
{
public:

virtual bool ProcessRequest(XrdSsiRequest *reqP,
                            unsigned short tOut=0);

virtual void RequestFinished(      XrdSsiRequest  *rqstP,
                             const XrdSsiRespInfo &rInfo,
                                   bool      cancel=true);

virtual bool Unprovision(bool forced=false);

             mySession(const char *sname)
                      : XrdSsiSession(strdup(sname)),
                        XrdSsiResponder(this, (void *)0) {…}
protected:
virtual     ~mySession{} {if (sessName) free(sessName);}
};
```

The above code shows a sample derivation. You must implement

- **ProcessRequest()**,
- **RequestFinished()** and
- **Unprovision()**

Since the sample session class will also be responding to requests it has inherited the **XrdSsiResponder** class. There are no methods in that class that you need to implement but you must tell the class that you inherited it by passing your **this** pointer to its constructor. The second parameter is usually zero but can be used to uniquely identify the object type and is useful if more than one type of object inherits the responder. See the header file for details.

The **ProcessRequest()** method called when a new client request is received. The **RequestFinished()** method is called after all of the response is sent to the client. The **Unprovision()** method is called when the session object is no longer needed because either the client called **Unprovision()** on the corresponding client-side session or the connection to the client was lost. In the event the connection was lost, **RequestFinished()** is called for each outstanding request indicating that it was cancelled before **Unprovision()** is called. This avoids having to keep track of every request.

3.4 Step 4: Provision a Resource

When a client provisions a resource by calling **XrdSsiService::Provision()** the **SSI** frameworks connects to a server that provides the resource. An equivalent resource object is created with added information that describes the client. That information is embodied in the **XrdSsiEntity** object pointed to by the **XrdSsiService::Resource** object. You may use the information for any purpose deemed necessary. The **SSI** framework then calls the service's **Provision()** method.

If provisioning is a fast operation, it can be done synchronously. Otherwise, you should hand off the operation to a separate thread and return. Whether or not the operation is synchronous, when the provision has completed, regardless of success, you must call the **ProvisionDone()** method in the resource object passed to you. If the provisioning succeeded you must provide the **XrdSsiSession** object that can be used as the target for client requests relative to that resource. If provisioning failed you must note the reason of the failure in the resource object's **eInfo** object and call **ProvidionDone()** with a nil pointer. When calling **ProvisionDone** with a nil pointer you also have the option to redirect the provisioning to another server or have the client retry at a later time. See the **XrdSsiService.hh** file for details.

The following code snippet shows an example of synchronous provisioning using the session class previously shown.

```
#include "XrdSsi/XrdSsiService.hh"
#include "XrdSsi/XrdSsiSession.hh"

void myService::Provision(XrdSsiService::Resource *resP,
                          unsigned short           timeout)
{
mySession *sessP = new mySession(resP->rName);

// Tell the resource it has been provisioned
//
resP->ProvisionDone(sessP);
}
```

Unlike the client-side provision, the server is given additional options on how a provisioning failure is treated. Two other options are available:

- redirect the client to another end-point where it will attempt to provision the resource, or
- delay the client for some specific amount of time before it attempts to provision the resource at the same end-point.

The following code snippet shows how this can be done.

```
#include "XrdSsi/XrdSsiService.hh"
#include "XrdSsi/XrdSsiSession.hh"

void myService::Provision(XrdSsiService::Resource *resP,
                          unsigned short           timeout)
{
static const int   delayTime = 10;
static const int   port      = 1094;
static const char *altHost   = "somehost.domain.edu";
       mySession  *sessP     = 0;

// If we are too busy then delay the client
//
     if (toobusy) resP->eInfo.Set("Too busy, try later!",
                                  EBUSY, delayTime);

// If we need to send the client elsewhere tell the caller
//
else if (nothere) resP->eInfo.Set(altHost, EAGAIN, port);

// OK we will provision and get a session object
//
else mySession *sessP = new mySession(resP->rName);

// Tell the resource we are done
//
resP->ProvisionDone(sessP);
}
```

3.4.1 Resource Names

While resource names are arbitrary, by default they must start with **/tmp**; which is rather useless. You control the form of valid resource names using the **all.export** configuration directive. This directive is essentially *mandatory* in order to be able to provision resources. The directive allows you to specify the leading characters of a valid resource name. There can be any number of these directives. For instance,

```
all.export /resource/ nolock r/w
```

Specifies that a valid resource must start with the sequence "/resource/" followed by a sequence of characters that includes Names are restricted to the following set of characters:

- Letters (upper or lower case),
- Digits (0-9), and
- Special characters: `!@#%^_-+=:./`

In general, paths may not contain shell meta-characters or imbedded spaces. Be aware that you must always specify the **nolock** and **r/w** options in the order shown; otherwise, resource provisioning fails.

It is possible to lift most restrictions on resource names by specifying the following configuration directive:

```
all.export * nolock r/w
```

Resource names are not checked for validity but still may not contain imbedded blank characters. If your **SSI** plug-in also is able to accept optional **CGI** information appended to the resource name (e.g. resourcename?var1=val[&var2=val[…]]) then you should specify

```
all.export *? nolock r/w
```

The added question makes **XRootD** scan for a question mark and split the resource into a name and a **CGI** string. See the configuration section for full details.

3.5 Step 5: Process a Request

Requests are always executed in the context of a session. When a client provisions a resource the session object created by the server handling the resource is effectively exported to the client. So, when a client calls **ProcessRequest()** on that exported session object the same call is made on the corresponding server's session object. This implements remote object execution. A following code snippet shows what needs to be generally done.

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiResponder.hh"

void mySession::ProcessRequest(XrdSsiRequest *reqP,
                               unsigned short tOut}
{
char *reqData;
int   reqDLen;

// Tell our responder to bind the request to this session
//
BindRequest(reqP, this);

// Get the request data buffer and its length
//
reqData = reqP->GetRequest(reqDLen);

// Process the request
//
•••
}
```

One of the first things you must do is to call **BindRequest()**. This tells the inherited responder object to record which session object that is responsible for handling the request. If the request will be handed off to another object (e.g. an agent object) that will respond to the request then that object's responder must be passed **BindRequest()** as the third parameter. This allows you to processes multiple requests using the same session object and is discussed in detail in the next section.

You then get the request data by calling **GetRequest()**. This can be done any number of times, as needed. To reduce memory consumption, call the request's **RelRequestBuffer()** when you no longer need to look at the request data. That method frees the memory used for the request data. So, make sure you no longer reference the request data buffer.

When the request is finished, successfully or not, a response must be posted via the inherited responder. The responder makes sure that the response is sent to the client and that the associated session object's **RequestFinished()** method is called when the client-server exchange has completed.

3.5.1 Multiple Simultaneous Requests

The **SSI** framework allows you to handle multiple requests within the context of a single session. Generally, a client can send another request to your session object at any time before its **Unprovision()** method is called unless your client is implemented in a way that prohibits it.

Allowing multiple simultaneous requests within a single session avoids all of the overhead of creating a session object (i.e. provisioning) and tearing it down (i.e. unprovisioning). This may be substantial depending on the service being provided.

If your session object is already capable of handling multiple requests you need not read further. This section assumes that your session object can only handle one request at a time.

There are two general ways that this can be done. The first is straight-forward and is based on creating sub-session. It works if your session object is relatively simple. The second approach uses session agent objects. These objects are responsible for actually handling the request but otherwise work within the context of the original session. The two approaches are described in the following sections.

It is important to remember that each call to a session object's **ProcessRequest()** is invoked on a new thread. This means there is no need for creating another thread to process the request in parallel. This simplifies thread management.

3.5.1.1 Simultaneous Requests Using Sub-Sessions

Assuming that your session object can handle one request at a time, there is no practical reason why you cannot simply create a new session object and hand off the request to it. The initial session object is simply the base object that handles request hand-off and unprovisioning. New session objects are considered sub-sessions that do all the heavy work. Below is a code snippet that shows how it can be done.

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiResponder.hh"

void mySession::ProcessRequest(XrdSsiRequest *reqP,
                               unsigned short tOut}
{
mySession *subSession = new mySession(…);

// Tell our responder to bind the request to the sub-session
//
BindRequest(reqP, subSession);

// Call the actual processing method. The sub-session should
// delete itself prior to returning. Maintaining a sub-session
// pool of objects can eliminate new-delete overhead.
//
subSession->ProcessIt(reqP, tout);
}

void mySession::ProcessIt(XrdSsiRequest *reqP,
                          unsigned short tOut}
{
char *reqData;
int   reqDLen;

// Get the request data buffer and its length
//
reqData = reqP->GetRequest(reqDLen);

// Process the request
//
•••

// Delete ourselves before returning
//
delete this;
}
```

The sub-session object is the same as a session object but has access to its own set of members, if any. The new additional method, **ProcessIt()**, can be private and is the method that does all the work.

There are two important things to keep in mind:

- The bound sub-session object's **RequestFinished()** method is called when the request object's **Finished()** method is called. So, the sub-session is responsible for cleaning up request resources.
- The base session object's **Unprovision()** is called when the either the client calls **Unprovision()** on its corresponding session object or the connection is lost. In either case, you are guaranteed that sub-session **RequestFinished()** is called for each outstanding request prior to the **Unprovision()** call in the base session object.

Sub-sessions work best if the session object is relatively simple and does not contain many management related members. Alternatively, you could have different session objects (e.g. one for the main session and another for a sub-session). However, using session agents may produce a better design because you need not implement all of the virtual session methods. In any case, using different session object derivations is similar to using session agents in approach.

3.5.1.2 Simultaneous Requests Using Session Agents

When it is cumbersome to spawn sub-sessions, the **SSI** framework allows you to hand off a request to session agent objects. A session agent is any kind of object you define to process a request. It is possible to have different kinds of agent objects and differentiate between them. The agent object must inherit the **XrdSsiResponder** class. This allows the agent to respond to the request and allows the request object to keep track of which agent is handling the request. This is important since there is only one session object and it is responsible for keeping track of all outstanding requests and their agents.

The **SSI** framework simplifies this management overhead by allowing you to determine which agent is handling which request. So, you only need to keep track of agent objects and perform the proper clean-up when the base session object's **RequestFinished()** is called for any particular request (the request object's **nextRequest** and **prevRequest** members can be used for this purpose).

A sample code snippet on how to use agent objects is shown on the next page.

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiResponder.hh"

class myAgent : public XrdSsiResponder
{
public:

void Execute();

     myAgent(XrdSsiRequest *reqP)
             : XrdSsiResponder(this, (void *)0),
               myRequest(reqP) {…}
    ~myAgent(){…}
private:
XrdSsiRequest *myRequest;
};

void mySession::ProcessRequest(XrdSsiRequest *reqP,
                               unsigned short tOut}
{
myAgent *theAgent = new myAgent(reqP);

// Tell our responder to bind the request to the agent
//
BindRequest(reqP, this, (XrdSsiResponder *)theAgent);

// Call the actual processing method. The agent should delete
// itself prior to returning. Maintaining an agent pool of
// objects can eliminate new-delete overhead.
//
theAgent->Execute();
}

void myAgent::Execute()
{
int   reqDLen;
char *reqData = myRequest->GetRequest(reqDLen); // Get request

// Process the request
//
•••

// Delete ourselves before returning
//
delete this;
}
```

Notice that the code snippet is very similar to the one used in the sub-session example. The difference here is that

- there is only one base session responsible for all requests,
- you need not implement any session methods in the agent object, and
- the request is bound to the base session and the agent's responder via **BindRequest()** using the third argument.

You could have used different session object derivations and bound the sub-session's responder in exactly the same way if you wish to use a sub-session approach. However, using agents may yield a simpler and more understandable design. For instance, it may allow you to only have agent objects inherit the **XrdSsiResponder** class and agent objects need not inherit the **XrdSsiSession** class. This provides a clean separation of function.

3.5.1.3 Keeping Track Of Different Responder Derivations

The responder object allows you to assign a derived class type when you inherit it via its constructor. This is handy if you bind different kind of objects to the request using **BindRequest()**. Since call requires that you up cast any object that inherited **XrdSsiResponder** to that class, you essentially loose that differentiation. The derived class type value can be used to down cast a responder object to the appropriate class in a type-safe way.

There are two ways you can assign a derived class type via the responder. The first is using a unique void pointer, shown below.

```
#include "XrdSsi/XrdSsiResponder.hh"

class myObject1 : public XrdSsiResponder
{
public:

     myObject1() : XrdSsiResponder(this,(void *)&type_Object1)
                  {…}
    ~myObject1(){…}

static char type_Object1;
};

char myObject1::type_Object1 = 'x'; // At file level
```

The member **type_Object1** is unique but common to all instances of **myObject1**. You can use the value of its address to verify that the derived class is in fact the desired one and safely down cast the responder object to **myObject1** using a **static_cast<>** operation.

Recall that **BindRequest()** ties a session and, optionally, a responder to a particular request. The request object allows you to get the pointers to the objects the request is bound to via its **GetSession()** and **GetResponder()** methods. In turn, the responder object allows you to extract the object type identifier of the object that inherited the responder using the responder's **GetObject()** method. The code snippet below shows how you can safely down cast a responder object to its derived class given a request pointer.

```
#include "XrdSsi/XrdSsiResponder.hh"

// reqP is a passed XrdSsiRequest pointer
//
myObject1 *obj1_P = 0;
XrdSsiResponder *theResponder = reqP->GetResponder();

// Check to make sure the request was actually bound. If it
// it was, get the full object pointer and its type. Using
// that information, do the appropriate down cast.
//
if (!theResponder) {handle condition}
   else {void *objType;
         void *objPntr = theResponder->GetObject(objType);
         if (objType == (void *)&myObject1::typeObject1)
            obj1_P = static_cast<myObject1 *>(objPntr);
            else • • •
        }

// Verify that we successfully down cast the responder
//
if (!objP) {handle condition}
```

The second way of uniquely identifying an object is to use an **enum** value. The **enum** would enumerate your object types. The responder also allows you to pass an arbitrary integer argument along with the type value; as shown in the next snippet as a third argument to its constructor. Pick either way but always use only one.

```
#include "XrdSsi/XrdSsiResponder.hh"

enum ObjectType {type_Object1, type_Object2, …};

class myObject1 : public XrdSsiResponder
{
public:

     myObject1() : XrdSsiResponder(this,(int)type_Object)
                   {…}
    ~myObject1(){…}
};
```

The subsequent code snippet shows how to use the **enum** method to appropriately down cast the responder object.

```
#include "XrdSsi/XrdSsiResponder.hh"

// reqP is a passed XrdSsiRequest pointer
//
myObject1 *obj1_P = 0;
XrdSsiResponder *theResponder = reqP->GetResponder();

// Check to make sure the request was actually bound. If it
// it was, get the full object pointer and its type. Using
// that information, do the appropriate down cast.
//
if (theResponder)
   {ObjectType objType;
    int objTArg; // Value passed as 3rd arg to the responder
    void *objPntr =
               theResponder->GetObject((int)objType, objTArg);
    switch(objType)
          {case typeObject1:
                obj1_P = static_cast<myObject1 *>(objPntr);
                break;
           • • •
          };
    };

// Verify that we successfully down cast the responder
//
if (!objP) {handle condition}
```

3.5.2 Request Processing and Threads

Since the **SSI** framework keeps track of all outstanding requests, you do not need to do so. However, you will likely need to or, at least keep track of sub-sessions or agent objects, if you want to relinquish the thread. This may be necessary if processing requires a lot of wait time (e.g. reading data from a slow device). Remember that each request is dispatched on a new thread. This works if the thread is kept constantly active or the processing is relatively short. Otherwise, you will likely need to implement processing queues and handle request processing as appropriate using a smaller pool of threads.

You can always find out which session and agent is bound to a request via the request's **GetSession()** and **GetResponder()** methods. This described in more detail in the section that discusses finishing a request.

3.6 Step 6: Post the response

After you process the request you must respond with a result. A response can be

- a data buffer along with the length,
- an error message with an errno value,
- an open file with a size whose contents are the response, or
- a stream object that supplies data.

The response is always contained in the **XrdSsiRespInfo** object, a private member in the **XrdSsiRequest** object. That **XrdSsiRespInfo** object is passed to your implementation of **RequestFinished()** so that you can release the response resources after the response has been sent to the client. The following table lists all the possible response types you may receive relative to the object's member contents.

rType is	**buff**	**blen**	**eMsg**	**eNum**	**fdnum**	**fsize**	**strmP**
isData	→ data	data length	n/a	n/a			
isError			→ message	errno code			n/a
isFile	n/a	n/a	n/a	n/a	file descriptor	file size	n/a
isStream	n/a	n/a	n/a	n/a			→ XrdSsiStream object

All responses are set by calling the appropriate **SetResponse()** method that an object inherited from the **XrdSsiResponder** class. Once a response has been posted to a request it cannot be changed. Hence, only a single response is allowed. An error response is shown in the following code snippet

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiResponder.hh"

•••

// Tell the client the request was not valid
//
SetErrResponse("Invalid request!", EINVAL);

•••
```

3.6.1 Sending Optional Metadata Ahead of the Response

The **SSI** framework allows a server to send metadata ahead of the response data. This metadata may be used for any purpose. For instance, the metadata may describe the response so that the client can handle it in the most optimum way.

You send metadata with the **XrdSsiResponder::SetMetadata()** method using the inherited **XrdSsiResponder** class. The metadata must be set before you post a response calling **XrdSsiResponder::SetResponse()**. The maximum amount of metadata that may be sent is defined by **XrdSsiResponder:: MaxMetaDataSZ** constant member.

The metadata buffer must remain persistent until the **RequestFinished()** method is called in the session object that has been bound to the request via a **BindRequest()** call.

3.6.2 Sending a Metadata-Only Response

The **SSI** framework allows a server to send only metadata as a response (i.e. the response only consists of metadata). This is rarely useful for sending short responses and avoiding additional client-server handshakes to determine that there is no actual response data other than the metadata. This is because the **SSI** framework automatically sends metadata and the actual data response, if it's a data buffer, if the length of the metadata plus the data buffer is less than or equal to the static constant **XrdSsiResponder:: MaxDirectXfr**. So, you can send short responses as data instead of metadata. That said, you may wish to use metadata-only responses for possible consistency reasons.

Be aware that metadata is always sent and the data buffer portion of the response may be converted to a stream response it is too large. This is done to minimize client-side memory usage.

You send metadata as outlined in the previous section. To send only metadata use the **SetNilResponse()** method to force metadata to be sent with an indication that there is no other response data. On the client side this will appear as a zero-length data buffer response. This equivalent to calling **SetResponse((const char *)0, 0)**.

3.7 Step 7: Finish or Cancel the Request

When the client-server exchange completes, the **SSI** framework calls the **RequestFinished()** method in the session object that has been bound to the request via a **BindRequest()** call. This allows you to release all resources dedicated to the request. A client-server exchange completes when either

- All of the response data has been sent to the client, or
- the client has cancelled the request before the full response has been sent.

A client cancels a request when it

- calls its request object's **Finished()** method before all of the response data has been sent,
- closes the TCP connection before all of the response data has been sent, or
- indicates in the **Finished()** call that it wishes to cancel the request.

In practice, client actions are asynchronous to the server. The server-side framework always calls **RequestFinished()** when the full response has been sent to the client irrespective of the client calling its own request's **Finished()** method. This is done to speed up client-server interactions. While practically consistent, it is not logically consistent because while the data is in transit to the client the client may cancel the request at the same time. The client-side framework makes it appear that the request was cancelled even though the server-side framework will not be aware of it.

The following code snipped shows the typical processing sequence for the **RequestFinished()** method.

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiSession.hh"

void SessionObj::RequestFinished(      XrdSsiRequest  *rqstP,
                                 const XrdSsiRespInfo &rInfo,
                                       bool            cancel)
{
•••
if (cancel) {stop processing the request if still active}

// Reclaim resources dedicated to the request
•••
```

Be aware that the object whose **RequestFinished()** is called is always determined by the session object pointer passed to responder's **BindRequest()** call.

If you are using sub-sessions or agents and need access to the your sub-session or agent object, you can easily do so in the **RequestFinished()** method. This allows you to recycle the agent or sub-session as needed without having to keep track of it. The following code snippet shows how it can be done.

```
#include "XrdSsi/XrdSsiRequest.hh"
#include "XrdSsi/XrdSsiSession.hh"
@include "myAgent.hh"

void SessionObj::RequestFinished(      XrdSsiRequest  *rqstP,
                                 const XrdSsiRespInfo &rInfo,
                                       bool            cancel)
{
XrdSsiResponder *theResponder = reqP->GetResponder();
myAgent         *agentP = 0;

// The responder can be any object that inherited the
// XrdSsiResponder class. It is set via BindRequest().
// Generally, it is a sub-session or agent if used at all.
// Below we get the actual object pointer
//
void *objType;
void *objPntr = theResponder->GetObject(objType);

// If the type identifier is the one we expect (it should be)
// we safely down cast it to the actual object that inherited
// that class, using a void pointer type identifier.
//
if (objType == (void *)&myAgent::typeObject1)
   agentP = static_cast<myAgent *>(objPntr);
   else {handle unexpected condition}

// If the request was cancelled but is still active the tell
// the agent assigned to the request to stop processing.
//
if (cancel) agentP->StopProcessing();

// Reclaim resources dedicated to the request
•••
```

The above code snippet uses agents and the type identification system provided by the SSI framework to easily recover the underlying object assigned to the request. See the section on agent derivations for complete details.

3.8 Step 8: Unprovision the Resource

A resource is un-provisioned via a call to the session object's **Unprovision()** method. This happens when the client

- calls the **Unprovision()** method in its corresponding session object or
- closes its **TCP** connection.

The **SSI** framework keeps track of all requests that are in progress for each provisioned resource. Prior to calling the base session's **Unprovision()** method it first calls **RequestFinished()** for each unfinished request indicating cancellation. This alleviates the need for the session object to keep track of in-progress requests. Afterwards, it calls **Unprovision()**. An argument to **Unprovision()** is set to indicate whether this is a normal client **Unprovision()** call or the un-provisioning was caused by the loss of the client's **TCP** connection (i.e. a forced un-provisioning). The client-side framework prohibits the client from un-provisioning a resource that has active requests. This forces the client to first cancel all of its session requests before calling **Unprovision()**. However, loss of the **TCP** connection forces request cancellation and subsequently un-provisioning all of the resources provisioned by the client.

The following code snippet shows what **Unprovision()** needs to do.

```
#include "XrdSsi/XrdSsiSession.hh"

void SessionObj::Unprovision(bool forced)
{
•••

// Reclaim resources dedicated to this session

•••
```

Be aware that the **Unprovisoion()** is called using the session object returned by the service object's **Provision()** method and is logically independent from any session object set via the responder's **BindRequest()** call. That said, the session object identified in the **BindRequest()** call is used for the **RequestFinished()** call; which may be the same object returned by **Provision()**.

3.9 Deleting Server-Side Objects

The following rules apply to safely delete the server-side objects described in the previous sections.

3.9.1 XrdSsiService Object

Technically, the **XrdSsiService** object can only be deleted in its **Stop()** method. However **SSI** framework never tries to stop a service.

3.9.2 XrdSsiService::Resource Object

The resource object may be deleted after its **ProvisionDone()** method is called by the service object. Therefore, you cannot keep a pointer to this object after provisioning completes.

3.9.3 XrdSsiSession Object

The **XrdSsiSession** object can only be deleted in its **Unprovision()** method. The **SSI** framework calls this method when the client call Unprovision() on its corresponding session object or the client's **TCP** connection is lost.

3.9.4 XrdSsiRequest Object

The request object is deleted after its **Finished()** method has been called. This, in turns, causes the **RequestFinished()** method to be called in the session object bound to the request via a **BindRequest()** call. You may not have a pointer to the request object after returning from **RequestFinished()**.

3.10 Overall Flow Summary

The table below shows the overall flow with a simple single data response. The flow is more complicated when the response is a data stream. Details on presented in the sections on streams.

Client Application	**Client-Side SSI Framework**		**Server-Side SSI Framework**	**Server-Side Service**
XrdSsiProvidorClient-> GettService()	Return XrdSsiService Object		XrdSsiProvidorServer-> GetService()	Return XrdSsiService object
Call Provision() *async return*	open() Call ProvisionDone() with XrdSsiSession Object	→	Call Provision()	Call ProvisionDone() with XrdSsiSession object
Create XrdSsiRequest Call ProcessRequest() *async return*	BindRequest() GetRequest() write()	→	read() Create XrdSSiRequest Call ProcessRequest()	BindRequest() GetRequest() Perform service
	Call ProcessResponse()	←	Notify client of pending response	Call SetResponse() As data response
Call GetResponseData() *async return*	read() Call ProcessResponseData()	↔	write() Call Finished()	Cleanup
Call Finished()	Cleanup			
Call Unprovision()	close() Recycle XrdSsiSession	→	Call Unprovision()	Recycle XrdSsiSession

4 XrdSsiStream

The **XrdSsiStream** object is used to provide data from a streaming source. Examples of streaming sources are:

- Socket
- Incremental database query
- Character device

Essentially, a streaming source is any data source that provides data in an incremental fashion whose size is not easily determined or, if it is known, whose size is too large to conveniently handle in one interaction with the data source.

The **XrdSsiStream** is an abstract class whose creator must provide a concrete implementation suitable to the data source in question. Any data source can be used but the implementation must adhere to the invocation restrictions defined herein.

The **XrdSsiStream** object supports two types of streamlining modes::

1) Active streams where, when asked for data, the object supplies the buffer containing the data, and
2) Passive streams where, when asked for data, a buffer is supplied by the requestor that is to be filled with data.

The two types of streaming modes are described in subsequent sections.

4.1 Relationship to other classes

The **XrdSsiStream** object is only meaningful in the context of an **XrdSsiRequest** object to affect a response. Hence, the **XrdSsiStream** object *is* the response to a request. Subsequently, the requestor obtains actual response data using the provided **XrdSsiStream** object.

Recall that responses are posted to a request object using the **XrdSsiResponder::SetResponse()** method. This means that any object wishing to post a stream response must

a) inherit the **XrdSsiResponder** class, and
b) be bound to the request with a previous **XrdSsiResponder::Bind()** invocation.

The above two requirements allow an object to process and respond to a request without any extensive request bookkeeping and maintains a 1-to-1 relationship between a request and the primary object responsible for responding to the request.

Typically, an **XrdSsiSession** object binds to a request but, in practice, any object that has inherited the **XrdSsiResponder** class may bind to a particular request. This allows multiple requests to be processed by a single session object.

4.2 Object Persistence

A stream object may not be deleted until all references to the object have been dropped. An explicit reference occurs when a stream object is exported via a call to **XrdSsiResponder::SetResponse()**. Only when **XrdSsiSession**::**RequestFinished()** is called with a pointer to the request object whose response holds the stream object can the creator of the stream object be assured that no **SSI** framework references exist to that stream object. Only then can the stream object be deleted.

The **XrdSsiSession::RequestFinished()** method is called when the **XrdSsiRequest::Finished()** is called indicating that the request has completed. A request can be marked as completed at any time even when not all of the stream data has been consumed. The stream implementation is responsible for reclaiming any unconsumed non-exported resources.

Be aware that active stream buffers that have been exported via the **XrdSsiStream::GetBuff()** method may only be reclaimed in the **XrdSsiStream::Buffer::Recycle()** method. This is because a reference to the buffer remains active until the buffer is recycled whether or not the request has been marked as finished.

4.3 Streaming Request-Response Sequence

The following diagram shows the sequence and time relationships between client actions and server actions relative to a stream. The sequence is the same whether an *active* or *passive* stream is used. The differences are described under the specific discussion of each type of stream. Red arrows show sequential calling sequences while block colors indicate individual threads. A detailed explanation follows.

Request-Response Sequence Using a Stream

Referencing the above diagram, the following steps occur:

1) The client constructs a request encapsulated by the **XrdSsiRequest** object pointed to by **crP** and passes it to the client-side session pointed to by **csP** for processing. The session object serializes the request object and sends it to its corresponding session object at some server. None of these actions block. If the request cannot be sent immediately it is queued and sent in the background.
2) When the call returns the application is free to perform any other required tasks. The request object will be informed that a response is ready when the server actually responds. In the mean time the server receives the request and recreates the request object pointed to by **srP**.
3) The **SSI** framework spawns a new thread to process the request. This is to avoid any timeout issues should the request take a long time to complete. It calls the server-side session object pointed to by **ssP** to process the request. In this example, the session object will be processing the request so it binds to the request to setup a 1-to-1 mapping between the request and the request processor.
4) The request is processed.
5) At this point the appropriate type of stream is created. If the results are either too large or cannot be immediately obtained an *active* stream may be created. Otherwise, a *passive* stream may be created. In either case **xsP** point to a stream of appropriate type.
6) Since the stream is the actual response, the session object calls **SetResponse()** pointing to the stream object. It can do so because it must have inherited the **XrdSsiResponder** class. As part of the **SetResponse()** call the **SSI** framework sends notification to the client that a response has been posted to the request object and returns. This is not blocking action.
7) At this point the session object, if using an *active* stream, is free to start filling a buffer with response data while the notification is in transit. At some point the client-side **SSI** framework receives the notification.
8) The **SSI** framework creates a passive stream, pointed to by **psP**, to relay the data from the server to the client. The framework matches the notification to the corresponding request object, **crP**. It then sets the passive stream as the response for the request using **SetResponse()**.
9) As part of the **SetResponse()** process, the **ProcessResponse()** method in the request object is called to notify the request that a response is ready. In this example, the request's **ProcessResponse()** method obtains a buffer to hold the response data.

10) It then calls the request object's **GetResponseData()** data method to fill the buffer with response data. Note that the **GetResponseData()** method is a convenience function that actually uses the passive stream to obtain the data freeing the application from having to deal with actual streams. The application could have used the passive stream directly if it wanted to. When the passive stream is asked to fill the buffer it sends a request for data to the server. Again, this is a non-blocking call.
11) Upon return from **GetResponseData()** the application is free to perform any other required tasks until it is informed that the data has arrived. In the mean time, the server has received the request for data.
12) *Active stream*: The **SSI** framework calls the *active* stream's **GetBuff()** method to obtain response data. The *active* stream simply hands over a buffer that contains some amount of data. Since this buffer has been exported to the framework it cannot be touched or freed until the framework indicates it is through with the buffer by calling the buffer's **Recycle()** method.
13) *Passive stream*: The **SSI** framework calls the *passive* stream's **SetBuff()** method to obtain response data. The framework supplies the buffer and its size and the stream must fill the buffer to the extent possible.
14) If enough data exists in the buffer, the data is sent to the client. Otherwise, the **SSI** framework will ask for (*active* step 12) another buffer or (*passive* step 13) another buffer fill until it accumulates enough data to send, as determined by the amount of data the client wanted to receive (i.e. the size of the client's buffer) or the stream indicates that no more data exists. This is known as impedance matching and is described in subsequent sections. Note that for *active* streams the supplied buffer's **Recycle()** method may be called during this process when the framework no longer needs to use the buffer. When called, the buffer should either be deleted or reclaimed for future use. Additionally, the session may start to pre-fill a buffer to satisfy a future call to **GetBuff()** while the previous data is in transit to the client.
15) Using a new thread, the **SSI** framework matches the data with the request, **crP**, that asked for the data and reads the data into the buffer supplied by the client.
16) The framework invokes the request's **ProcessResponseData()** method. The application handles the data as needed and may repeat steps 9 and 10 if it needs more data. In any case, it must return to the caller.

17) At some point, the request is finished and the application invokes the request's **Finished()** method. If not all of the data has been received by the **SSI** framework, the server is notified to discard any remaining data. In any case, the framework removes all references to the request object which allows the application to delete the object. The request object may not be deleted until its **Finished()** method is called. On the server side, one of two poSSIbilities exist, as follows:
    a) If not all of the data has been transmitted to the client but a finished request is received, then for *active* streams the server-side **SSI** framework calls **Recycle()** on any buffers it still has control over. For all stream types, the framework invokes the corresponding request's **Finished()** method indicating that the request was cancelled.
    b) If the stream indicated that no more data exists and the remaining data has been sent to the client, then for *active* streams **SSI** framework calls **Recycle()** on any buffers it still has control over. For all stream types, the framework invokes the corresponding request's **Finished()** method indicating that the request was successfully completed.

    In either case, it invokes request object's **Finished()** method which invokes the **RequestFinished()** method of the session object bound to the request. That method can reclaim the stream object. Active streams are assured that either all buffers exported by the stream's **GetBuff()** method have already been recycled or will be upon return.

4.3.1 Active Streams

An active stream object is one that provides a buffer containing the data when requested to provide data. This means that the implementation of an active stream is responsible for buffer management. Active streams only make sense for a service provider (i.e. the server-side plug-in). Active streams are never created by the **SSI** client-side framework. A sample active stream derivation is shown below.

```
#include "XrdSsi/XrdSsiStream.hh"

class myStream : public XrdSsiStream
{public:

Buffer  *GetBuff(XrdSsiErrInfo &eInfo, int &dlen, bool &last);
•••
          myStream() : XrdSsiStream(XrdSsiStream::isActive) {…}
virtual ~myStream() {…}
•••
};
```

4.3.1.1 Active Stream Impedance Matching

Active streams pose a particular challenge since the **SSI** framework does not know how much data the active stream will deliver. Unless there is some indication in the request-response protocol, the client may not even know how much response data to expect. Consequently, clients typical provide a buffer of some size and ask the **SSI** framework to fill it to the extent possible.

When the server-side **SSI** framework initially receives a read request for the size equivalent to the size of the client's buffer, it asks the active stream for a buffer passing it the number of bytes needed to completely satisfy the read request. If the stream provides fewer bytes, the **SSI** framework subtracts the number of bytes returned from the original size, recycles the current buffer, and asks for another buffer of reduced size so that it can completely satisfy the request. This repeats until the either the request is satisfied or the stream indicates that no more data remains.

If the active stream provides more data that is needed, the amount needed is used to satisfy the request and the remaining data is held (i.e. the buffer is not recycled). The data in the buffer is used to satisfy a subsequent request for data.

Subsequent requests for data either start anew or use data in an existing but to the extent possible. In this way, the server-side **SSI** framework matches unequal buffer sizes (i.e. impedance matches). Clearly, the most efficient transfer mode is where the active stream provides exactly the amount of data needed.

Impedance matching does not come without a latency cost. If the stream is slow in supplying data and the client's buffer is large then the client waits until the full buffer can be filled. This may cause the client to timeout and cancel the request. In the end, how an active stream supplies data is a trade-off between its speed and the amount of data requested by the client at one time. While small requests may be used to better match the speed of a stream they come at the cost of more client-server interactions which incur additional network latency.

The good solution for slow streams is to pre-fill buffers ahead of client requests. The assumption is that it is rare that a request will be cancelled so as wasting the effort. The preceding stream diagram illustrates the appropriate places in the request-response sequence where pre-filling can be done with good results. Typically, two or three ready buffers should suffice.

The natural question is whether the initial buffer should be ready before posting the stream as a response to the request object. The answer really depends on the speed of the stream. If it is slower than the client turn-around then at least one buffer should be ready before posting the stream response. Otherwise, there is likely enough time to pre-fill at least one buffer afterwards. The general solution to this problem is to implement a consumer-produce algorithm with a fixed number of buffers.

4.3.2 Passive Streams

A passive stream is a stream that fills a caller supplied buffer. Passive stream need not manage any buffers as the caller always supplies one. Inherently, they are simpler to implement and provide the user of the stream complete freedom on how to manage the buffer space. Because of their simplicity and generality, only passive streams are used by the client-side **SSI** framework.

While passive streams provide much of the same capabilities as active stream they have one significant drawback; buffers cannot be pre-filled by a stream without significant data movement which, many times, is very expensive in terms of CPU utilization. Frequent and large movements of data also incur a large amount of memory contention that may cause significant latency in overall processing. Thus passive streams should only be used for data sources that can provide (i.e. fill a buffer) at reasonably high speeds. A typical derivation is shown below.

```
#include "XrdSsi/XrdSsiStream.hh"

class myStream : public XrdSsiStream
{public:

bool     SetBuff(XrdSsiRequest *rqstP, char *buff, int blen);
•••
         myStream() : XrdSsiStream(XrdSsiStream::isPassive)
                      {…}
virtual ~myStream() {…}
•••
};
```

4.3.2.1 Passive Stream Impedance Matching

While passive streams are relatively simple, the **SSI** framework still attempts to impedance match data availability from the stream with the amount of data requested by the client.

When the server-side **SSI** framework initially receives a read request for the size equivalent to the size of the client's buffer, it asks the passive stream to completely fill buffer by passing it the number of bytes needed to completely satisfy the read request. If the stream provides fewer bytes, the **SSI** framework subtracts the number of bytes provided from the original size, and asks for another buffer fill of reduced size so that it can completely satisfy the request. This repeats until the either the request is satisfied or the stream indicates that no more data remains.

Impedance matching does not come without a latency cost. If the stream is slow in supplying data and the client's buffer is large then the client waits until the full buffer can be filled. This may cause the client to timeout and cancel the request. In the end, how an active stream supplies data is a trade-off between its speed and the amount of data requested by the client at one time. While small requests may be used to better match the speed of a stream they come at the cost of more client-server interactions which incur additional network latency.

Because of the simplicity of passive stream, there is no efficient way to pre-fill buffers without incurring excessive memory-to-memory data movement which may cause excessive CPU usage or memory access latency.

5 Clustering SSI Servers

Since **SSI** executes in the **XRootD** framework, all **SSI** servers can be clustered using the native **XRootD** clustering services via the **cmsd** daemon. An **XRootD** cluster uses a manager node (also known as the redirector) to manage up to 64 server nodes. If your configuration has more than 64 servers, these are handled by additional daemons called supervisor nodes. Each supervisor can handle up to 64 nodes and there is no limit on the number of supervisors you may have. The details of **XRootD** clustering can be found in the Cluster Management Service Configuration Reference. In this chapter we focus on a simple clustered configuration and what needs to be done by your **SSI** server-side implementation to support it.

First, you need to understand how the **cmsd** locates resources in a cluster. The basic steps are as follows:

- A client connects to the **XRootD** daemon (hitherto called **xrootd**) that pairs with a manager **cmsd**. Here the **xrootd** handles data traffic while the **cmsd** is responsible for locating resources and monitoring the health of a cluster.
- The client then provisions a resource by sending the request to the manager **xrootd**. Since the manager **xrootd** knows it is in a clustered environment; it asks its corresponding **cmsd** to locate the resource to be provisioned.
- If the **cmsd** has not seen the resource before, it sends a query to every **SSI** server node that potentially has the resource asking one or more to respond affirmatively if it can provide that resource.
- If no **SSI** servers respond then the client is told that the resource is not available.
- If one or more **SSI** servers respond that they have the resource, the **cmsd** chooses one of the **SSI** servers and tells the **xrootd** where to redirect the client to provision the resource.
- The client then reconnects to the server that has the resource and asks the server to provision the resource. All subsequent interactions relative to the resource are directed to that server.

The above scenario means that the **cmsd** on the **SSI** server needs ask your plug in whether or not the resource is available using some kind of interface. This is done using the **QueryResource()** method in the **XrdSsiProvider** class. The steps that need to be taken are very similar to those required to create a provider for the **xrootd** daemon. The next section describes what needs to be done for the **cmsd**.

5.1 Define the Service Provider for the cmsd

Server-side services are provided via the **XrdSsiProvider** object. While the client-side has one that is built-in the server-side must define one. There are actually two types of providers:

- A **cmsd** provider that needs only to ascertain whether or not a resource is available on the node on which its **QueryResource()** method is called, and
- A **xrootd** provider that actually provides the service via the **GetService()** method. The provider still needs to be able to ascertain resource availability.

The lookup provider is used by **cmsd** to ascertain the availability of a resource on a particular server node. Since the **cmsd** never actually provides any service it *never* obtains a service object from the provider object. The lookup provider object is pointed to by the global pointer **XrdSsiProviderLookup** which you must define and set at library load time (i.e. it is a file level global static symbol).

The service provider is used by **xrootd** to actually process client requests. This was described in a previous section. While it can be the same service provider object; in most cases it makes sense to have two different objects provide the **QueryResource()** method as it usually it differs in actual implementation.

When your library is loaded, the **cmsd** locates the provider using the symbol **XrdSsiProviderLookup**. Initialization fails if the symbol cannot be found or if its value is nil.

The two methods that you must implement in your derived **XrdSsiProvider** object for the **cmsd** are:

- **Init()**, and
- **QueryResource()**

When a client provisions a resource, and the resource has not been used before, all servers indicating that they can provide the resource are asked via this method if they can actually provide the resource. The **SSI** framework then picks one of those servers to handle the client's requests relative to the provisioned resource. This effectively implements a real-time dynamic resource registration system.

The following code snippet shows how you would typically define the provider pointer at file level.

```
#include "XrdSsi/XrdSsiProvider.hh"

class MyLookupProvider : public XrdSsiProvider {•••};

XrdSsiProvider *XrdSsiProviderLookup = new MyLookupProvider;
```

Once the provider object is found, its **Init()** method is called. The method should initialize the object for its intended use. Below is a sample derivation of an **XrdSsiProvider** class for the **cmsd**.

```
#include "XrdSsi/XrdSsiProvider.hh"
#include "XrdSsi/XrdSsiService.hh"

class myLookUpProvider : public XrdSsiProvider
{
public:

// Init() is always called before any other method
//
bool           Init(XrdSsiLogger *logP,  XrdSsiCluster *clsP,
                    const char   *cfgFn, const char    *parms,
                    int           argc,  char         **argv

                   ) {••• initOK = true; // If all went well
                      return initOK;
                     }

// The QueryResource() method determine resource availability
//
XrdSsiProvider::
rStat          QueryResource(const char *rName,
                             const char *contact=0
                            );

               myLookUpProvider() : initOK(false);
virtual       ~myLookUpProvider() {}

private:

bool initOK;
};
```

6 Configuration

In order to use the server-side **SSI** framework you must configure an **XRootD** server. There are two modes: unclustered (i.e. stand-alone single server) and clustered (i.e. multiple servers clustered by a redirector or manager node). There is no need to configure anything on the client's side. The client side program simply links against **libXrdSsi.so** to obtain access to the client-side **SSI** framework.

The following sections show the minimal set[1] of **XRootD** directives needed to use the **SSI** framework for each type of configuration. Finally, the list of **SSI**-specific directives is explained. These may help you customize the **SSI** framework.

6.1 Unclustered XRootD SSI Configuration

```
# Tell XRootD to use only the SSI framework. If you wish to
# also use the filesystem features of XRootD then add the
# keyword default (i.e. xrootd.fslib libXrdSsi.so default).
#
xrootd.fslib libXrdSsi.so

# Turn off async processing as this does not work with SSI
#
xrootd.async off

# Declare the valid prefix for resource names. You can have
# as many of these directives as you wish, each specifying a
# different prefix (substitute the actual prefix for respfx).
# If you wish to use resource names without a leading slash,
# read the section describing resource names.
#
all.export respfx nolock r/w

# Specify the resource lookup function to be used.
#
oss.statlib -2 libXrdSsi.so

# Specify the location of the shared library implementing
# you SSI service. See the SSI svclib directive for details.
#
ssi.svclib libpath
```

[1] For a complete list of directives see the relevant reference manuals on **XRootD** and **cmsd**.

6.2 Clustered XRootD SSI Configuration

In addition to the directives specified for an unclustered configuration, you need to specify the additional directives shown below.

```
# Tell XRootD who the cluster manager is (a.k.a. redirector).
# Substitute for manhost the fully qualified DNS name of the
# node running the cluster manager. For manport the port
# number that it should use to listen for requests. Do so
# everywhere you see manhost and manport here.
#
all.manager manhost:manport

# Assign the appropriate role to SSI servers and the cluster
# manager. This is done using an if-else-fi clause. In this
# way the same configuration file can be used everywhere.
#
if manhost
all.role manager
else
all.role server
fi
```

6.3 Configuring SSI and XRootD Resource Names

Recall that the **all.export** directive tells the **SSI** framework how to process resource names. It is likely you will need to specify this directive to establish a coherent naming convention for your resources. While there are numerous options only two are meaningful for the SSI framework, **nolock** and **r/w**; as previously described.

While resource names are typically used to identify **SSI** specific resources, the **SSI** framework allows you to also define traditional file system resources that can be used by clients using the same server. This is enabled using the **ssi.fspath** directive described uner the **SSI** specific directives.

When you wish to enable an **xrootd** server to handle **SSI** requests and file system request, you need to alter the **xrootd.fslib** directive, add **ssi.fspath** directives, and the appropriate **all.export** directives, as shown below.

```
# Tell XRootD to handle SSI requests as well as regular XRootD
# file system requests (notice the addition of default).
#
xrootd.fslib libXrdSsi.so default

# Tell the SSI framework which resource name prefixes refer
# to actual file system requests. Substitute for fspfx a
# file system path prefix. There can be many ssi.fspath
# directives.
#
ssi.fspath fspfx

# Declare the valid prefix for resource names. You can have
# as many of these directives as you wish, each specifying a
# different prefix. Substitute the actual prefix for respfx
# You must also export the file system prefixes declared via
# the ssi.fspath directive. However, you must not specify the
# nolock option for these to prevent file corruption.
#
all.export respfx nolock r/w

all.export fspfx appropriate_options

# Continue with the directives specified in the previous
# sections for unclustered or clustered configurations.
```

6.4 SSI Specific Directives

All **SSI**-specific directives start with the prefix "**ssi.**" to differentiate them from other types of directives.

6.4.1 fspath

```
ssi.fspath fspfx
```

Function

Specify the resource name prefix that routes an **SSI** request to the **XRootD** file system plug-in.

Parameters

fspfx The resource name prefix, when seen, is to route the request to the file system plug-in instead of the **SSI** service.

Default

There is no default, see the notes for more information.

Notes

1) If you enable the **XRootD** file system plug-in to function alongside the **SSI** framework then you need to specify which resource name prefixes actually refer to the file system and not the **SSI** service. When a resource name with the matching prefix is encountered, the request is routed to the file system plug-in.
2) Each *fspfx* specified with the **fspath** directive must also be specified using the all.export directive. However, you must *not* specify the **nolock** option for these exports.

Example

```
ssi.fspath /tmp
all.export /tmp
```

6.4.2 opts

```
ssi.opts [auth aval] [maxrsz rsz[k | m | g]]
         [requests rnum] [respwt sec]
aval: 0 | 1 | 2
```

Function

Specify the resource name prefix that routes an **SSI** request to the **XRootD** file system plug-in.

Parameters

auth *aval*

Specifies how client authentication information is passed to the SSI service. Specify for *aval*:

0 - do not supply any authentication information (default).

1 - only supply the client's host name.

2 – supply full authentication information as configured.

maxrsz *rsz*

Specifies the maximum size of a valid request. Specify for *rsz* the largest possible request size. The *rsz* can be suffixed by **k**, **m**, or **g** to indicate **k**ilo-, **m**ega-, or **g**iga-bytes; respectively. The default is 2m.

requests *rval*

Specifies the maximum number of request objects to hold in reserve for future requests. Specify for *rval* a number greater than 0, the default is 256.

respwt *sec*

Specifies the number of seconds the client should be asked to wait for a response when a response is not ready. The *sec* can be suffixed by **d**, **h**, or **m**, or **s** to indicate **d**ays, **m**inutes, or **h**ours; respectively. The default is 24855d.

Default

```
ssi.opts auth 0 maxrsz 2m requests 256 respwt 24855d
```

Example

```
ssi.opts requests 512
```

6.4.3 svclib (*required*)

```
ssi.svclib lib [parms]
```

Function

Specify the shared library that implements the SSI service.

Parameters

lib The path to the shared library that contains the code the implements the protocol.

parms Parameters to be passed to the service initialization method at load time.

Default

There is no default; this is a required directive.

Example

```
ssi.svclib libService.so
```

6.4.4 trace

```
ssi.trace [-]option

option:   {all | debug | off} [[-]option]
```

Function

Specify execution tracing options.

Parameters

option Specifies the tracing level. One or more options may be specified. The specifications are cumulative and processed left to right. Each option may be optionally prefixed by a minus sign to turn off the setting. Valid options are:

all	selects all possible trace levels other than debug
debug	adds additional tracing for debugging purposes
off	traces nothing

Defaults

Tracing is disabled.

Notes

1) All tracing is forcibly enabled when the daemon is invoked with the **–d** option.
2) All previous trace settings are discarded when **off** is encountered.

Example

```
ssi.trace all
```

7 Managing Resources in a Cluster

When the **XrdSsiProvider::Init()** method is called in a clustered environment, it is passed a pointer to an **XrdSsiCluster** object. This object is used to manage resources within cluster. It is critical that resources be properly managed for efficient execution of client requests in the cluster. The two main actions using the cluster object are:

1. Register or unregister the presence of a resource name at a node.
2. Request suspension or resumption of service.

7.1 Registering and Unregistering Resource Names

Recall that when the cmsd looks nodes that can provide a resource name that it has not yet seen, it asks each node whether or not it has the particular resource. Nodes responding that they have the resource are eligible to receive client requests for that resource. This mechanism is used to automatically register (affirmative response) and unregister (no response) resource names relative to the set of nodes in the cluster. All of this is driven by calling **XrdSsiProvider::QueryResource()**.

Since automatic registration is specific to a point in time, the cluster object has methods that must be used by a node to inform the cluster manager of changes in the registration status of its resources. Specifically,

Added(const char ****name*, bool** *pend***=false)**

must be called whenever a new resource becomes available or has changed its pending status since the last time the node was queried about the resource name. Since it is inconvenient to track whether or not a node was asked about the *name*, the **Added()** method may be called at any time whether or not the node was queried about the resource. This allows registrations to be current and accurate.

Removed(const char ****name*)**

must be called whenever a new resource becomes unavailable since the last time the node was queried about the resource name. Since it is inconvenient to track whether or not a node was asked about the *name*, the **Removed()** method may be called at any time whether or not the node was queried about the resource. This allows registrations to be current and accurate.

Failure to properly use these two methods may cause clients to be sent to the wrong nodes or being told a resource does not exist even when it does. While the former is auto-correcting with an added latency cost the latter is not correctable and the client will normally declare a fatal error.

7.2 Suspending and Resuming Service

A node has control of whether or not it is willing to accept client requests. At initial start-up the cluster manager assumes all nodes in the cluster can accept client requests unless told otherwise. A node can control it status by using the following **XrdSsiCluster** methods:

Suspend(bool *perm***=false)**

may be called to prohibit clients from being sent to the issuing node in the future. When perm is true, this status persists across node restarts. Otherwise, the suspension is cancelled after a server at the node restarts.

Resume(bool *perm***=true)**

may be called to resume service after a previous Suspend() call. When called with perm equal to true, any permanent suspension undone. Passing a value of false, maintains the previous suspension status upon server restart.

Permanent suspensions are meant to easily implement maintenance mode. Should a server's node enter maintenance the server can be permanently suspended to make sure that an inadvertent restart does not resume service before the completion of maintenance.

Since suspensions are time driven events, it is possible that client requests have been directed to the server before the suspension is acknowledged by the cluster manager. So, it is normal to still get several requests after calling **Suspend()**. In this case, the server may redirect these requests back to the cluster manager for reprocessing.

Suspend() and **Resume()** can be used to manage the load at the server. Three special helper methods are available to make it easier to do so. These **XrdSsiCluster** methods are based on the concept of unit of service and are used as follows:

Resource(int *n*)

is used to declare *n* arbitrary units of available service. While it is normally called once, it can be called any number of times. Each invocation returns the resource unit value that was previously established.

Reserve(int *n*=1)

is used to declare that the server has undertaken a task that will use *n* units of service. The default is one unit. The amount is deducted from the available service units and should the remainder fall below 1; **Suspend()** is automatically called to temporarily suspend service.

Release(int *n*=1)

is used to declare that the server has completed a task that required the *n* units of service. The default is one unit. The amount is added to the available service units. If the amount available transitions from a non-positive amount to a positive value; **Resume()** is automatically called to allow new client requests to be sent to the server.

8 Starting the SSI Server

Starting an **SSI** server is equivalent to starting an xrootd in stand-alone node or an xrootd-cmsd pair in a clustered node. There are numerous command line options for starting each type of server. Refer to the **Xrd/Xroot**d and **cmsd** reference manuals for details on these options. With few exceptions, they are the same for **xrootd** and **cmsd** daemons. There is one common option that deserves special treatment here. The option allows you to pass command line arguments to the **XrdSsiProvider:: Init()** method. When starting a server-side daemon you may specify command line arguments to be passed to the **SSI** plug-in as follows.

{**xrootd** | **cmsd**} [*options*] [*arguments*] **-+xrdssi** *options_arguments*

The *options_arguments* tokens are passed to the **XrdSsiProvider::Init()** method via the *argv* parameter with *argc* set to the actual number of arguments. The **argv[0]** token is the same as the one passed to the daemon (i.e. executable path). The *options_arguments* tokens end either at the end of the command line or when another "**-+**" option is encountered. The **-+xrdssi** option must appear after all of the **xrootd/cmsd** specific options and arguments have been specified on the command line.

9 Document Change History

9 Aug 2014

- Document produced.

29 Jul 2015

- Numerous changes and additions.

7 Sep 2015

- Describe how to send and receive metadata.

28 Sep 2015

- Describe metadata-only responses.
- Describe direct transmission of data buffer responses and client-side processing of such responses.
- Document the **opts** directive **respwt** option.
- Add section on request timeouts.

11 Oct 2015

- Remove references to the **mwfiles** option as this is no longer needed.

3 Jun 2016

- Document **XrdSsiRequest::SSRun()** method.
- Document how to pace responses via return value from **ProcessResponseData()** callback.
- Document how to restart suspended callbacks using **XrdSsiRequest::RetsartDataResponse()**.