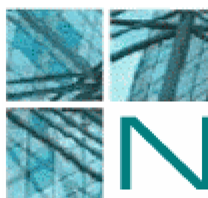

Technical Report NEESgrid-2003-09
www.neesgrid.org

(Whitepaper Version: 1.1)
Last modified: May 8, 2003)

Proposed Design for NEESgrid Telepresence Referral and Streaming Data Services

Carl Kesselman[1], Laura Pearlman[1], Gaurang Mehta[1]

[1] University of Southern California Information Sciences Institute, Marina del Rey, CA 90292

Feedback on this document should be directed to neesgrid-si@neesgrid.org

Acknowledgment: This work was supported primarily by the George E. Brown, Jr. Network for Earthquake Engineering Simulation (NEES) Program of the National Science Foundation under Award Number CMS-0117853.

Table of Contents

1 Introduction 3
2 The NEESgrid Streaming Data Service 5
2.1 Protocols Used by the NSDS Server 6
2.1.1 Communication between the NSDS Server and Clients: NSDS Requests 6
2.1.2 Communication between the NSDS Server and Clients: NSDP Data 7
2.1.3 Communication between the NSDS Server and Local DAQs 7
2.2 NSDS Server Architecture 8
2.2.1 Overview 8
2.2.2 Modules 9
2.2.3 I/O and Authentication Module 9
2.2.4 NSDP Parsing Module 9
2.2.5 Client Channel Management Module 9
2.2.6 Authorization Module 9
2.2.7 Subscription Module 10
2.2.8 The Data Streaming Module 10
2.2.9 The DAQ Communication Module 10
2.2.10 Threading 11
3 The NEESgrid Telepresence Referral Service 11
3.1 The NTRS Protocol 12
3.2 NTRS Server Architecture 13
3.3 Access Control 13
Appendix: Status as of May, 2003 14
Acknowledgments 19

1 Introduction

A typical earthquake engineering experiment today (Figure 1) involves at least one data acquisition (DAQ) and control system, zero or more *control points* (shake tables, actuators, etc.) and one or more *sensors* (accelerometers, strain gauges, etc). A DAQ receives data from sensors; a typical experiment might involve one combination DAQ/control system that sends control messages to a shake table (or to a small number of actuators) and receives data from several sensors, and a second DAQ that receives data from many additional sensors.

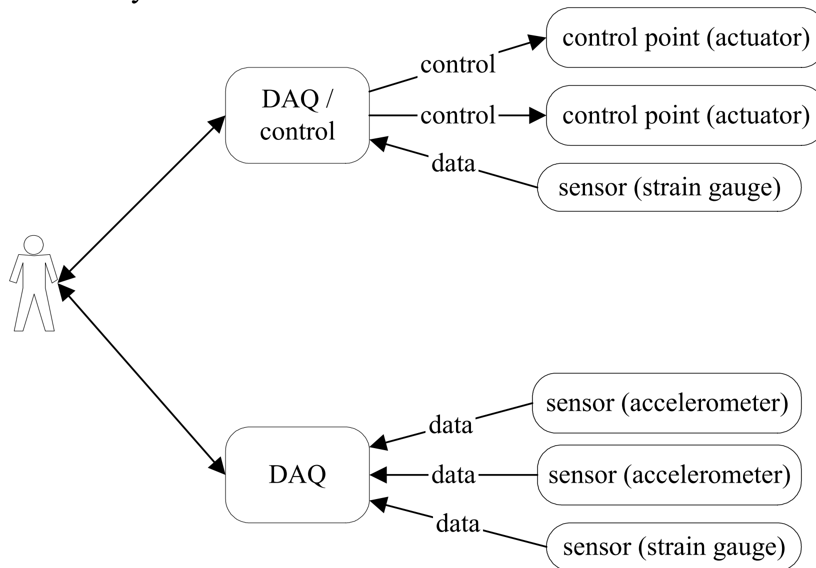

Figure 1: An experiment (without NEESgrid services)

The experiment is typically controlled by a person via the user interface provided by each DAQ and control system. This person can view sensor data via the user interface while the experiment is in progress, and export the data when the experiment is finished. In hybrid experiments, the physical experiment is controlled by a software simulation via an interface particular to the DAQ and control systems being used; the software simulation reads sensor data, makes calculations, and sends control requests back.

At the same time, researchers who are physically present at the site of an experiment can view its progress visually, either in person or via direct video connections. As part of the NEESgrid project, experiment sites will provide streaming video from some of their cameras during experiment runs.

The NEESgrid Streaming Data Service (NSDS) will provide a common protocol for use by remote applications to receive streaming data from a running experiment over NEESgrid. A companion service, NTCP, is used to send control requests to a running experiment over NEESgrid.*

* In earlier documents, the protocols supported by NSDS and NTCP were referred to collectively as NTOP. The NTCP service is described in a separate NEESgrid technical report.

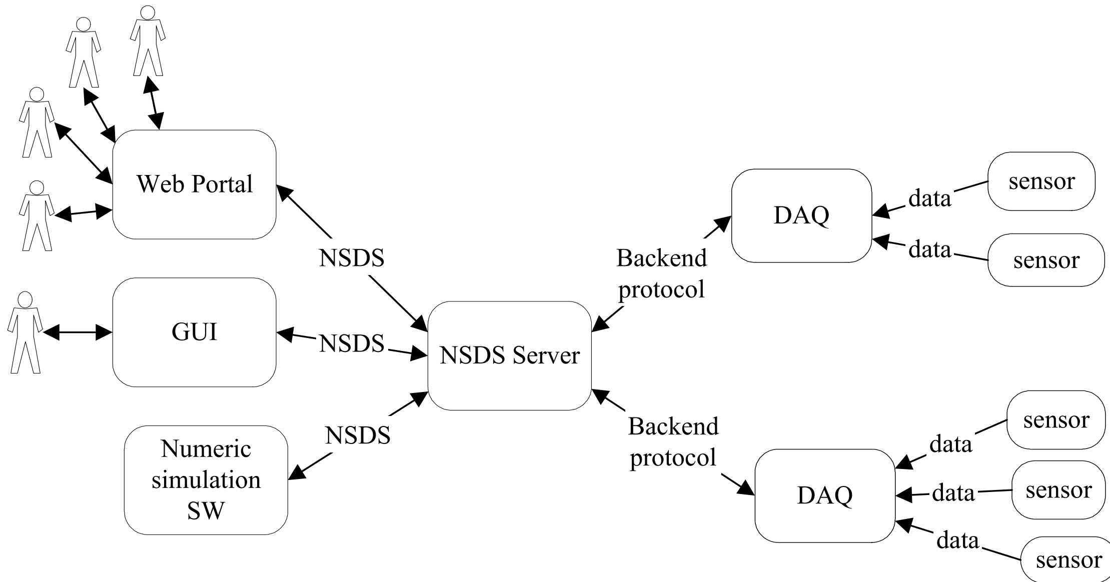

Figure 2: An experiment using an NSDS server

A second service, the NEESgrid Telepresence Referral Service (NTRS) will be used by remote applications to locate and obtain authorization for access to data and video services. In essence, the NTRS translates information (sensor names and authorization information) from the experiment domain into protocol domains.
.

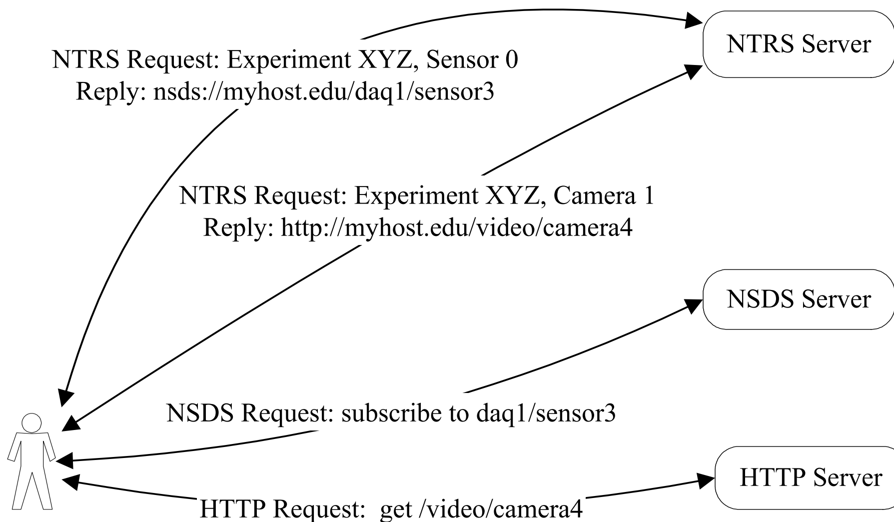

Figure 3: Using the NEESgrid Telepresence Referral Service

A remote user (or application) who wishes to receive streaming data (or video) may first contact the NTRS server at the site where the experiment is running to translate the logical names of sensors and cameras used in an experiment into URLs that can be used to contact streaming data or video servers and refer to those sensors or cameras. The NTRS server may also (depending on the implementation and protocols used) perform

additional actions to communicate authorization information to the streaming data or video servers.

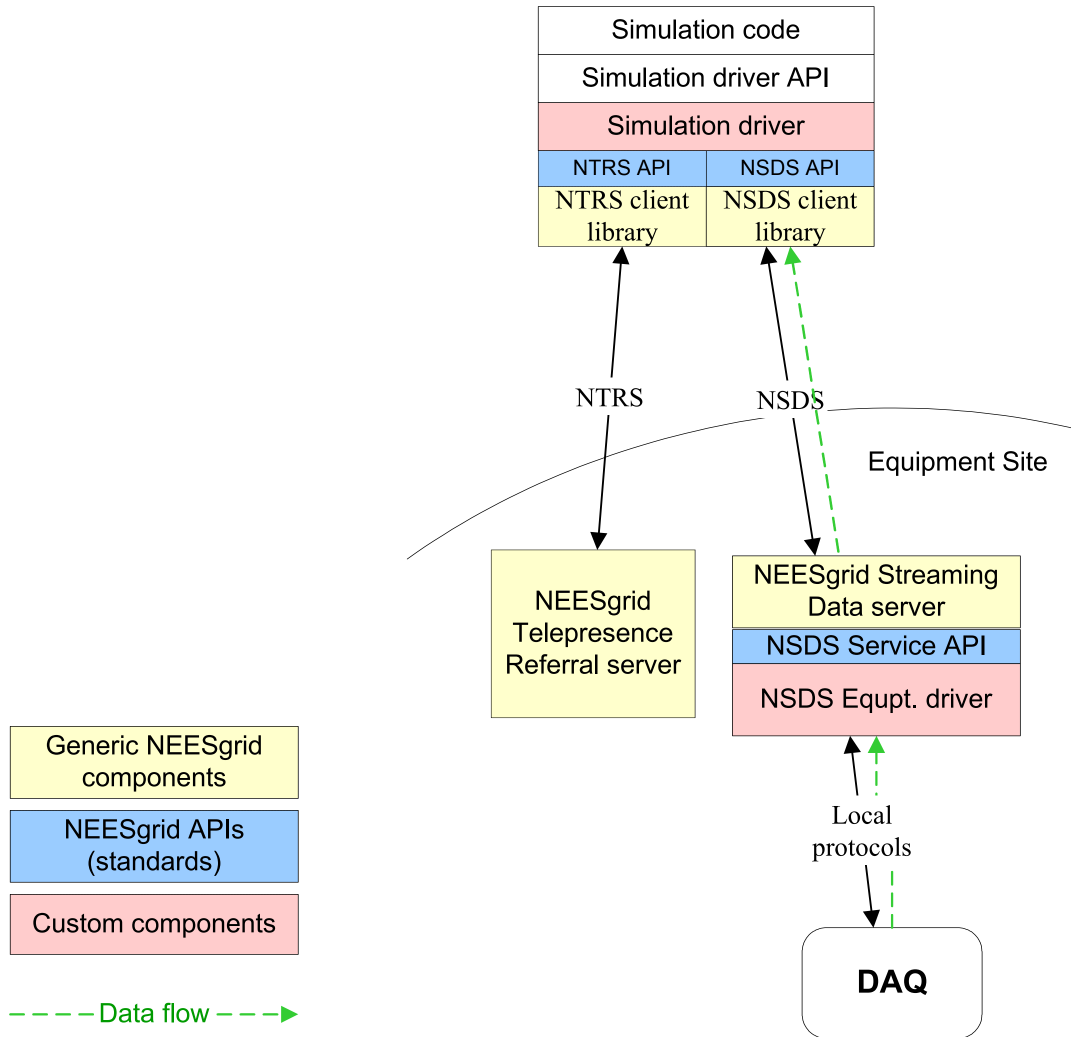

Figure 4: Software interfaces between NSDS, NTRS, DAQ systems, and applications.

2 The NEESgrid Streaming Data Service

The NSDS server will accept requests to subscribe to (and unsubscribe from) the streaming data for a sensor, enforcing access control policies for these requests. It will handle requests from multiple remote clients and subscriptions to multiple data streams simultaneously. A separate set of services, not described in this document, provide for the location of and access to data (including video data) after the experiment has concluded.

2.1 Protocols Used by the NSDS Server

2.1.1 Communication between the NSDS Server and Clients: NSDS Requests

The protocol used by NSDS should be relatively easy to parse and extend; for that reason we have chosen to implement it as a web service, using SOAP encoding over HTTP.

NSDS requests include:

- **Create_channel** (create a client data channel, to which sensor data may be sent):
  - request arguments: ip address and port number (both optional)
  - reply: a unique channel name (i.e., no two channels on the same NSDS server will have the same name). If an IP address and port number were specified, then the server opens a connection to that address/port, and the channel name refers to that connection. If no address/port was specified, then the channel name refers to the connection over which the **create_channel** request was sent.
  - access control requirements: *read* permission on any sensor.
- **Subscribe** (subscribe to a sensor data channel)
  - request arguments: sensor name, client channel name (as returned by create_channel)
  - reply: success or failure. If success, data from the named sensor is sent to the client channel until an *unsubscribe* request is received, the connection is closed, or the experiment trial is finished.
  - access control requirements: *read* permission on the sensor channel (access control is discussed in Section 3.3), and the request must come from the same entity that created the client channel.
  - other possible extensions:
    - decimation (results in only a sampling of data being sent to the listener channel),
    - other stop criteria (send at most N data points, stop at a particular time, etc.),
    - alternate output data formats,
    - virtual sensors -- features such as decimation or alternate output formats could be implemented by specifying different sensor names, rather than by adding extensions to the subscribe request format.
- **Unsubscribe** (cancel a subscription):
  - arguments: sensor name as in *subscribe* request
  - result: data from the named sensor will no longer be sent to the specified listener ip/port.
  - reply: success or failure
  - access control: must be same entity that did the associated *subscribe* request.
- **Close_channel** (close a client channel):
  - arguments: client channel name (as returned by create_channel)
  - result: the connection associated with the specified client channel is closed, any subscriptions referring to that channel are cancelled, and the client channel name will no longer be recognized by subscribe requests.
  - access control requirements: the request must be made by the entity that created the client channel.

    - possible extensions: client channels should be closed automatically if they do not have any subscriptions associated with them for some period of time.
- **Query sensor instantiation** (maybe?)
    - arguments: sensor name (as would be specified to the *subscribe* request)
    - result: true or false -- indication of whether there is a physical sensor associated with this name, and possibly more detailed information if the local DAQ supports this kind of query
    - access control: *query* permission on the sensor or control point
    - note: this request is probably not essential -- the metadata database should be used for basic information, such as whether a sensor is a strain gauge or an accelerometer.

2.1.2 Communication between the NSDS Server and Clients: NSDP Data

The format used for streaming will be different from the NSDS protocol itself; in fact, future implementations of the NSDS server will support alternate output formats.
The requirements for this data format are the following:

- It should include the information that client applications need, which would include at least the following for each data point:
    - the sensor name, to support applications that receive data from more than one sensor on one connection.
    - a timestamp (relative to the start of the trial), to support synchronization within a trial
    - the actual data value (in units of voltage -- unit conversion data should be available in the metadata database)
- It should be relatively easy to integrate with existing client applications (i.e., it should be relatively easy to parse).
- It should be compatible with the rest of the NSDS; that is, an application should be able to send NSDS requests and receive data on the same connection. This is not, however, an absolute requirement, as applications may use separate connections for their data channels.

2.1.3 Communication between the NSDS Server and Local DAQs

The protocols used to communicate between the NSDS server and the local DAQs may vary based on the capabilities of each local DAQ; it is, therefore, important to structure the NSDS server in such a way that it is relatively easy to install different drivers for different DAQs (and possibly to configure the NSDS server to use different drivers for different DAQs simultaneously). However, we will attempt to define a single protocol that will be supported by several of the most common DAQ systems, and will provide a driver for that protocol.

The requirements of this protocol are:

- It must support the streaming of sensor data from the DAQ to the NSDS server.
- It should include requests for establishing connections for sensors -- either in advance, before a trial begins, or (if supported by the DAQ) creating and shutting down connections as needed during the course of a trial.

- It should include requests to notify the NSDS server when a trial is finished (if supported by the DAQ).
- The DAQ side of this protocol should be relatively easy to implement on as many of the NEESgrid sites' existing DAQ implementations as possible.

2.2 NSDS Server Architecture

This section is intended to describe how an NSDS server may be implemented; this is not the only possible architecture for an NSDS server, and it is possible that the architecture eventually implemented may differ from what is described in this section.

2.2.1 Overview

The NSDS server may consist of these modules:

- The *i/o and authentication module* handles low-level network i/o and authentication (and parses any proxy restrictions present in the authentication credential); this is a standard Globus component.
- The *NSDS parsing module* parses an NSDS request.
- The *authorization module* determines whether or not an NTOP request is authorized, based on the local configuration and any proxy restrictions.
- The *subscription module* handles *subscribe* requests: it communicates with the *data streaming module* to locate (or create) the appropriate *input data stream* and *output data stream*, and creates a mapping between the two. It also handles *unsubscribe* requests in a similar fashion
- The *data streaming module* reads input data streams from the *DAQ communication module*, formats the data according to the NTOP protocol, and writes it out to the appropriate *output data streams*.
- The *administration* module handles local administration requests.

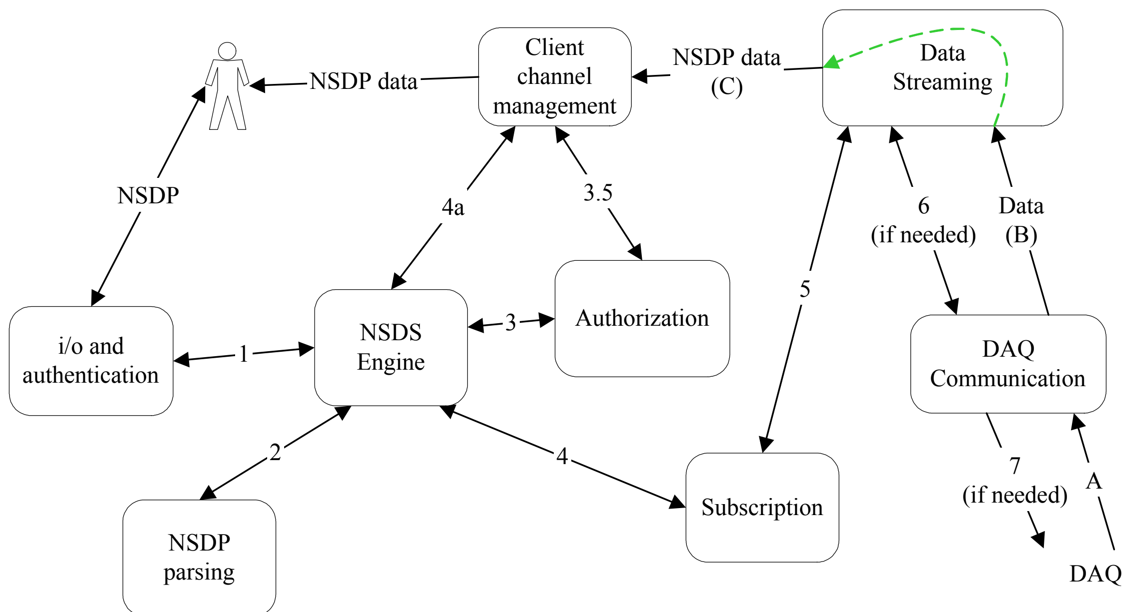

Figure 5: NSDS server modules

For example, a client channel creation request is read by the i/o and authentication module and passed to the NSDS engine (1), which sends it to the NSDP parsing module (2) for parsing, the authorization module (3) to check authorization, and then to the client channel management module (4a). The client channel management module then assigns a channel name to the connection (after creating the connection if appropriate) and returns the status to the NDSS engine, which then communicates the result to the client.

A subscription request goes through the same authentication, parsing, and authorization steps, and is then forwarded to the subscription module (4); in this case, however, the authorization module queries the client channel management module (3.5) to determine the owner of the client channel specified in the request. The subscription module locates the appropriate input data (sensor) channel, creates a mapping between the two streams, and returns the status to the NSDS engine, which then communicates the result to the client. If the requested sensor stream does not exist, the streaming data module contacts the DAQ communication module (6) to create one. Depending on the properties of the particular DAQ being used, the DAQ communication module may contact the DAQ itself (7) to initiate a sensor stream.

The DAQ communication module also reads data from the local DAQs; when data arrives (A), the DAQ communication module parses it and forwards it (as an input data stream) to the data streaming module (B). The data streaming module checks to see which output data channels are subscribed to that input stream, formats the data, and sends it, via the client channel management module, out to each waiting output channel.

2.2.2 Modules

2.2.3 I/O and Authentication Module

The i/o and authentication module performs raw i/o and authentication, including parsing any access restrictions in the authentication credential.

2.2.4 NSDP Parsing Module

The NSDP parsing module translates raw input into a data structure containing a parsed NSDP request.

2.2.5 Client Channel Management Module

The client channel management module handles communication with client data channels. When a channel is created, this module assigns a unique identifier to the channel, keeps track of the channel's owner, and opens the corresponding network connection (if appropriate). This module also writes data out to client channels and cleans up when a channel is closed (or when it detects that the underlying connection has been closed).

2.2.6 Authorization Module

The authorization module combines the NSDP request structure with the configured access control policy (including any policy from the authentication credential) and gives

a yes/no authorization decision. In some cases, this module communicates with the Client Channel Management Module to determine ownership of a client channel.

2.2.7 Subscription Module

The subscription module translates a subscribe or unsubscribe request into a request for the data streaming module and returns the result.

2.2.8 The Data Streaming Module

The Data Streaming module performs two basic functions:

1. It maintains mappings of:
   - input stream names (as expressed in NSDS requests) to input streams (data structures)
   - input streams to client data channels (subscriptions)
2. It reads data from the DAQ Communication module and forwards it out to clients.

It can be divided into two modules, a Data Stream Mapping Module and a Data Stream Flow Module. The Mapping module communicates with the Subscription Module (to handle subscribe and unsubscribe requests), answers "what output streams are subscribed to this input stream" queries from the Flow Module, and handles "remove all subscriptions to this input stream" requests from the DAQ Communication Module (these requests would be sent when an input stream closes down; e.g., when an experiment trial ends) and "remove all subscriptions involving this client data channel" requests from the Client Data Channel Module. The Flow module communicates with the DAQ Communication module (which sends it data) and with the Client Data Channel Module. [Note: it may make sense to subdivide the Flow module into submodules to do data formatting, decimation, etc.]

2.2.9 The DAQ Communication Module

The DAQ Communication Module performs these functions:

1. It handles requests from the Data Stream (Mapping) Module to create an input stream; that is, to accept a DAQ name and channel name as input and do whatever is appropriate to get input from that channel.
2. It reads and parses input streams and forwards that data to the Data Stream (Flow) module).
3. It notifies the Data Stream (Mapping) module when an input stream has closed.

The DAQ Communication Module is actually a thin wrapper around a DAQ Driver -- a DAQ-implementation-specific set of routines that handles communication with the DAQ. The DAQ Communication Module handles the mapping of the logical DAQ name and sensor name (e.g., "daq0, sensor 3" to physical DAQ and channel name (e.g., "daq1.mysite.edu, channel 5") and performs most of its functions by locating the appropriate driver for each sensor channel and calling routines from that driver.

Because DAQ drivers may be provided by third parties, it is important to have a clearly defined and published driver interface and to provide well-documented utility functions for DAQ driver authors to use. The initial DAQ driver will consist of these functions:

- open_sensor_channel (open a data channel to the DAQ)
  - arguments: physical sensor name
  - result: on success a sensor channel data structure corresponding to the opened sensor channel
- close_sensor_channel (close a DAQ sensor channel)
  - arguments: a sensor channel data structure
  - result: success/failure
- sensor_channel_read_loop (read a sensor stream from a DAQ)
  - arguments: a sensor channel data structure
  - result: this function should initiate a thread-safe loop that reads data from the sensor channel, convert it to a canonical format, and call a standard function to send the formatted data to the Streaming Data Module. If this function detects that the experiment has ended, it should call a standard function to notify the Streaming Data Module.

2.2.10 Threading

The NSDS server will almost certainly need to be threaded for performance. This raises questions about what level of consistency is required; we may need to consult with the user community for answers.

1. *Should we guarantee that all data sent to an output stream (possibly from several input streams) be sent in the order that it was read?* Probably not, as the data is timestamped.
2. *Should we guarantee that all data sent from one input stream to one output stream be sent in the order that it was read from the input stream?* Again, probably not, as the data is timestamped.
3. *Should we guarantee that all control requests sent from one client connection to one control point be forwarded to that control point in the order received?* Probably.
4. *Should we guarantee that all control requests sent from one client connection to any of the control points in an experiment be forwarded in the order received?* Again, probably.
5. *Should we guarantee that all control requests from all clients to one control point be forwarded in the order received?* How likely is it that more than one client will be sending control requests to the same control point during one experiment?

3 The NEESgrid Telepresence Referral Service

The NTRS server performs two functions:

- It translates names from the experiment domain into the protocol domain, i.e., for each sensor or camera that is located at the local site and used in a NEESgrid experiment, it translates the {experiment name, relative sensor or camera name} pair into a URL that can be used to refer to data from that sensor or camera, and

- It translates authorization information from the experiment domain into the protocol domain. If a remote user is authorized to read data from an {experiment name, relative sensor or camera name} pair, the NTRS server will attempt to ensure that the user can actually read that data from the local site -- by delegating an authorization credential, by performing dynamic access control operations on video servers, or by other means.

Some day-to-day administration tasks will need to be done by local site personnel. Each of these tasks could be accomplished via an administrative protocol, or by updating local configuration files (and either signaling the NTRS server to reread them, or having the server poll those files periodically). Of course, if an administrative protocol is chosen, the server should make sure that the configuration change will persist if the server is restarted (e.g., by writing the information to a database). These tasks include adding and deleting the {experiment name, sensor or camera name} to URL map entries.

3.1 The NTRS Protocol

This protocol consists of a single request:

- **translate_name**:
    - arguments: experiment name, sensor or camera name
    - returns: on success, a URL referring to that sensor or camera plus typed authorization information (e.g. a null value, a flag indicating whether or not the client should expect a delegated authorization credential, or some kind of authorization token).
    - access control requirements: *translate* permission on the sensor or camera; the authorization result of the request (delegated credentials, etc.) depend on the user's additional permissions on that sensor/camera.

3.2 NTRS Server Architecture

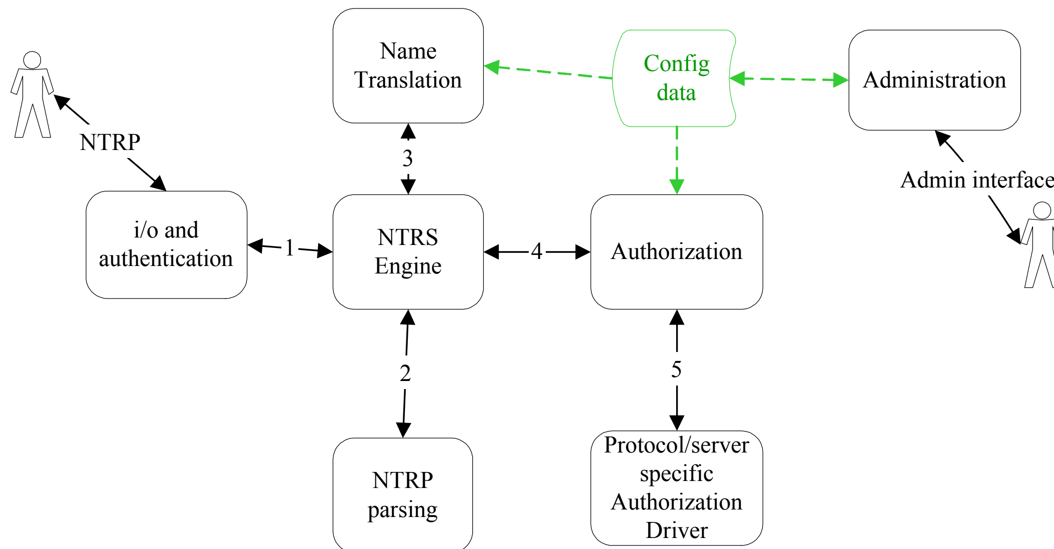

Figure 6: NTRS Server Architecture

An NTRP request is read by the i/o and authentication module and parsed by the NTRP Parsing module. The translation module handles the name translation (experiment/sensor to URL). The authorization module reads experiment-domain authorization information from its local configuration and combines that with any restrictions embedded in the authentication credentials, verifies that the remote user is allowed to access the requested {experiment, sensor or camera}, and passes that to an authorization driver (chosen based on the translated protocol and server), which performs the appropriate actions to ensure that the remote user will actually have access.

3.3 Access Control

Local site administrators will be able to grant permissions to perform the *translate* and *read* operations on experiment-domain sensors or cameras (such as "experiment xyz, sensor 1") and the *read* action (see section 2.1.1) on sensors using a local configuration file read by the NTRS server; these permission statements use the experiment-domain names of the sensors and cameras. Local site administrators will also be able to grant permissions to perform the *read* operation on logical sensors (such as "daq 1, sensor 3") on NSDS servers. If the administrators of an NSDS server grant *read* permissions on all local sensors to the NTRS server at that site, then the NTRS server can delegate a credential to a user that will allow that user to read a sensor. For example, if

- the NSDS server on *nsds.site.edu* is configured to give the NTRS server read access to all sensors, and
- the NTRS server is configured to
    - give Carol read permission on {experiment xyz, sensor 1} and
    - to map {experiment xyz, sensor 1} to "nsdp://nsds.site.edu/daq1/sensor3",

then Carol can send a translation request to the NTRS server and receive a credential that she can use to authenticate to the NSDS server and gain access to sensor3 on daq 1 on that server.

If the local site administrators grant the NTRS permissions to a CAS server, community administrators may grant these permissions to other community members. For example, a typical scenario might be:

1. Alice is designated as a site administrator at *site.edu*. She uses local configuration files on *nsds.site.edu* to grant, to the NTRS server at *nees..site.edu*, permission to read all sensors on the NSDS server at *nsds.site.edu*. She uses the local configuration file on *nees.site.edu* to grant, to the CAS server, permission to perform all actions on all sensors at her site. The central CAS administrators for the NEESgrid community grant her permission, within the CAS server, to create CAS resources under *nees.site.edu*.
2. Alice finds out that Bob is the PI for experiment XYZ, which will run on resources at site.edu. She uses CAS to:
   - create an object ("/nees.site.edu/XYZ") that represents that experiment at her site, and
   - grant permission to Bob to create CAS objects within that experiment (such as "/nees.site.edu/XYZ/sensor1") that represent sensors and control points within that experiment and to grant permissions on them.
3. Bob then either:
   - create CAS objects for each sensor and control point and use CAS to grant permissions on them ("this group of people can *translate* and *read* sensor /nees.site.edu/XYZ/sensor1"), or
   - grant permissions that span the entire experiment ("this group of people can read all the sensors in experiment /nees.site.edu/XZY").
4. In the meantime, Alice performs the other configuration tasks associated with setting up an experiment (e.g., at *nees.site.edu*, she creates the mappings that map "experiment XYZ, sensor 1" to "nsdp://nsdp.site.edu/daq1/sensor3").
5. If Bob has granted *read* and *translate* permissions to Carol on sensor1 for experiment XYZ, then Carol can contact the CAS server to obtain a credential which she can then use to connect to the NTRS server at *nees.site.edu* with those permissions. The NTRS server will perform the requested translation for her and delegate the appropriate credential; in this case, she will receive a credential that she can use to authenticate to the NSDS server at nsdp.site.edu and gain access to nsdp://nsdp.site.edu/daq1/sensor3.

Appendix: Status as of May, 2003

An initial NSDS implementation has been released as part of the NEES alpha-1 distribution, and was demonstrated at the NEES Awardees Meeting in November, 2002. This implementation provides very basic authorization capabilities and data formats. The following WSDL schema describes the NSDS service in its current implementation; we expect to make some extensions and minor changes to this schema in the coming year.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="NsdsDefinition"

targetNamespace="http://samples.ogsa.globus.org/nsds/nsds_port_type"

xmlns:tns="http://samples.ogsa.globus.org/nsds/nsds_port_type"
           xmlns="http://schemas.xmlsoap.org/wsdl/"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types>
  <xsd:schema
targetNamespace="http://samples.ogsa.globus.org/nsds/nsds_port_type"

xmlns:tns="http://samples.ogsa.globus.org/nsds/nsds_port_type"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema">

      <xsd:complexType name="NsdsChannelNameType">
       <xsd:sequence>
        <xsd:element name="channelname" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NsdsDaqNameType">
       <xsd:sequence>
        <xsd:element name="daqdriver" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NsdsSensorNameType">
       <xsd:sequence>
        <xsd:element name="sensor" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NsdsSubscribeReturnType">
       <xsd:sequence>
        <xsd:element name="result" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NsdsBooleanReturnType">
       <xsd:sequence>

        <xsd:element name="result" type="xsd:boolean"/>
       </xsd:sequence>
      </xsd:complexType>


      <xsd:complexType name="NsdsDaqStatusReturnType">
       <xsd:sequence>
        <xsd:element name="daq-status" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="NsdsSubscribeType">
```

```
        <xsd:sequence>
         <xsd:element name="experiment" type="xsd:string"/>
         <xsd:element name="sensor" type="xsd:string"/>
         <xsd:element name="channelname" type="xsd:string"/>
        </xsd:sequence>
       </xsd:complexType>

    <xsd:complexType name="NsdsAddTrialType">
        <xsd:sequence>
         <xsd:element name="trial" type="xsd:string"/>
         <xsd:element name="sensor" type="xsd:string"/>
         <xsd:element name="driverid" type="xsd:string"/>
         <xsd:element name="daq_port" type="xsd:string"/>
        </xsd:sequence>
   </xsd:complexType>

    <xsd:complexType name="NsdsRemoveTrialType">
        <xsd:sequence>
         <xsd:element name="trial" type="xsd:string"/>
         <xsd:element name="sensor" type="xsd:string"/>
        </xsd:sequence>
       </xsd:complexType>

    <xsd:complexType name="NsdsTrialNameType">
        <xsd:sequence>
         <xsd:element name="trial" type="xsd:string"/>
        </xsd:sequence>
       </xsd:complexType>

    <xsd:element name="createChannel">
      <xsd:complexType/>
    </xsd:element>
    <xsd:element name="createChannelResponse"
type="tns:NsdsChannelNameType"/>
    <xsd:element name="subscribe" type="tns:NsdsSubscribeType"/>
    <xsd:element name="subscribeResponse"
type="tns:NsdsSubscribeReturnType"/>
    <xsd:element name="unsubscribe" type="tns:NsdsSubscribeType"/>
    <xsd:element name="unsubscribeResponse"
type="tns:NsdsBooleanReturnType"/>
    <xsd:element name="closeChannel" type="tns:NsdsChannelNameType"/>
    <xsd:element name="closeChannelResponse"
type="tns:NsdsBooleanReturnType"/>
    <xsd:element name="queryDaq" type="tns:NsdsDaqNameType"/>
    <xsd:element name="queryDaqResponse"
type="tns:NsdsDaqStatusReturnType"/>
    <xsd:element name="addTrial" type="tns:NsdsAddTrialType"/>
    <xsd:element name="addTrialResponse"
type="tns:NsdsBooleanReturnType"/>
    <xsd:element name="removeTrial" type="tns:NsdsRemoveTrialType"/>
    <xsd:element name="removeTrialResponse"
type="tns:NsdsBooleanReturnType"/>

  <xsd:element name="flushFile">
      <xsd:complexType/>
    </xsd:element>
```

```
    <xsd:element name="flushFileResponse"
type="tns:NsdsBooleanReturnType"/>
    <xsd:element name="flushTrial" type="tns:NsdsTrialNameType"/>
    <xsd:element name="flushTrialResponse"
type="tns:NsdsBooleanReturnType"/>

  </xsd:schema>

</types>

<message name="CreateChannelInputMessage">
 <part name="parameters" element="tns:createChannel"/>
</message>
<message name="CreateChannelOutputMessage">
  <part name="parameters" element="tns:createChannelResponse"/>
</message>
<message name="SubscribeInputMessage">
  <part name="parameters" element="tns:subscribe"/>
</message>
<message name="SubscribeOutputMessage">
  <part name="parameters" element="tns:subscribeResponse"/>
</message>
<message name="UnsubscribeInputMessage">
  <part name="parameters" element="tns:unsubscribe"/>
</message>
<message name="UnsubscribeOutputMessage">
  <part name="parameters" element="tns:unsubscribeResponse"/>
</message>
<message name="CloseChannelInputMessage">
  <part name="parameters" element="tns:closeChannel"/>
</message>
<message name="CloseChannelOutputMessage">
  <part name="parameters" element="tns:closeChannelResponse"/>
</message>
<message name="QueryDaqInputMessage">
  <part name="parameters" element="tns:queryDaq"/>
</message>
<message name="QueryDaqOutputMessage">
  <part name="parameters" element="tns:queryDaqResponse"/>
</message>
<message name="AddTrialInputMessage">
  <part name="parameters" element="tns:addTrial"/>
</message>
<message name="AddTrialOutputMessage">
  <part name="parameters" element="tns:addTrialResponse"/>
</message>

<message name="RemoveTrialInputMessage">
  <part name="parameters" element="tns:removeTrial"/>
</message>
<message name="RemoveTrialOutputMessage">
  <part name="parameters" element="tns:removeTrialResponse"/>
</message>

<message name="FlushFileInputMessage">
  <part name="parameters" element="tns:flushFile"/>
</message>
```

```
<message name="FlushFileOutputMessage">
  <part name="parameters" element="tns:flushFileResponse"/>
</message>
<message name="FlushTrialInputMessage">
  <part name="parameters" element="tns:flushTrial"/>
</message>
<message name="FlushTrialOutputMessage">
  <part name="parameters" element="tns:flushTrialResponse"/>
</message>


<portType name="NsdsPortType">
  <operation name="createChannel">
    <input message="tns:CreateChannelInputMessage"/>
    <output message="tns:CreateChannelOutputMessage"/>
  </operation>
  <operation name="subscribe">
    <input message="tns:SubscribeInputMessage"/>
    <output message="tns:SubscribeOutputMessage"/>
  </operation>
  <operation name="unsubscribe">
    <input message="tns:UnsubscribeInputMessage"/>
    <output message="tns:UnsubscribeOutputMessage"/>
  </operation>
  <operation name="closeChannel">
    <input message="tns:CloseChannelInputMessage"/>
    <output message="tns:CloseChannelOutputMessage"/>
  </operation>
  <operation name="queryDaq">
    <input message="tns:QueryDaqInputMessage"/>
    <output message="tns:QueryDaqOutputMessage"/>
  </operation>

  <operation name="addTrial">
    <input message="tns:AddTrialInputMessage"/>
    <output message="tns:AddTrialOutputMessage"/>
  </operation>

  <operation name="removeTrial">
    <input message="tns:RemoveTrialInputMessage"/>
    <output message="tns:RemoveTrialOutputMessage"/>
  </operation>

  <operation name="flushFile">
    <input message="tns:FlushFileInputMessage"/>
    <output message="tns:FlushFileOutputMessage"/>
  </operation>

  <operation name="flushTrial">
    <input message="tns:FlushTrialInputMessage"/>
    <output message="tns:FlushTrialOutputMessage"/>
  </operation>

</portType>
</definitions>
```

```
The NTRS service has not yet been implemented; we plan to implement
this service, or provide equivalent functionality with an existing
service, within the next year.
```

Acknowledgments

This work was supported by the NSF NEESgrid project. We wish to acknowledge the contributions of other SI team members, particularly Nestor Zaluzec and Paul Hubbard.

.