



*Building the National Virtual Collaboratory
for Earthquake Engineering Research*

NEESgrid

Technical Report NEESgrid-2003-26

www.neesgrid.org

(Whitepaper Version: 1.0
Last modified September 15, 2004)

The NTCP Control Plugin

Laura Pearlman¹, Mike D'Arcy,¹ Pawel Plaszczak², Carl Kesselman¹

¹USC Information Sciences Institute, Marina del Rey, CA

²Argonne National Laboratory, Argonne, IL

Feedback on this document should be directed to laura@isi.edu

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.

<u>1</u>	<u>Introduction</u>	3
<u>2</u>	<u>Control Plugin Interface Definition (API)</u>	3
<u>2.1</u>	<u>Methods defined in the Control Plugin Interface</u>	3
<u>2.1.1</u>	<u>The propose method</u>	3
<u>2.1.2</u>	<u>The execute method</u>	4
<u>2.1.3</u>	<u>The cancel method</u>	4
<u>2.1.4</u>	<u>The getControlPoint method</u>	4
<u>2.1.5</u>	<u>The setParameter method</u>	5
<u>2.1.6</u>	<u>The getParameter method</u>	5
<u>2.1.7</u>	<u>The openSession method</u>	5
<u>2.1.8</u>	<u>The closeSession method</u>	5
<u>2.2</u>	<u>Types Used by the Control Plugin Interface</u>	6
<u>2.2.1</u>	<u>ControlPointType</u>	6
<u>2.2.2</u>	<u>ControlPointGeomParameterType</u>	6
<u>2.2.3</u>	<u>ControlPluginTransactionHandle</u>	7
<u>2.2.4</u>	<u>ParameterType</u>	8
<u>3</u>	<u>Control Plugin Use Cases</u>	8
<u>3.1</u>	<u>A direct hardware control server</u>	8
<u>3.2</u>	<u>A Remote Hardware Control Server</u>	9
<u>3.3</u>	<u>A Proxy Server</u>	9
<u>3.4</u>	<u>A Computational Simulation</u>	10
<u>3.5</u>	<u>An NTCP Gateway to a Simulation-Building Tool</u>	10
<u>3.6</u>	<u>Gateway to a Different Programming Language</u>	11
	<u>Acknowledgments</u>	11

1 Introduction

The NEESgrid Teleoperations Control Protocol (NTCP)¹ is used to perform remote control of physical experiments or simulations. An *NTCP control plugin* is a java class that implements the interface described in this document to communicate with a control system (or simulated control system). In our NTCP implementation, the NTCP server is configured to use a control plugin; when the server receives a request, it performs some “generic” NTCP operations (such as validating the request and updating transaction state) and then makes calls into the control plugin to carry out the request.

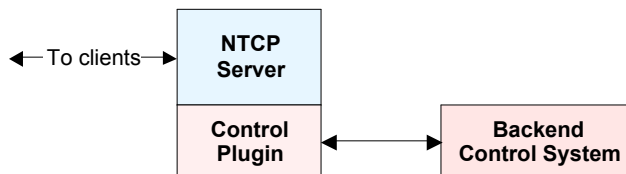


Figure 1: The NTCP server, control plugin, and backend control system.

NTCP plugins can be written to cause an NTCP server to act as:

- A server that controls hardware directly attached to the local system, or
- A server that controls hardware via some (non-NTCP) network protocol, or
- A proxy server that forwards requests to other NTCP servers, after applying policy and performing control point name mapping, or
- A computational simulation that accepts requests (and sends replies) via the NTCP protocol, or
- A gateway into a simulation development framework (such as Matlab).

These use cases are described later in this document.

2 Control Plugin Interface Definition (API)

The control plugin is a Java interface includes a method corresponding to each NTCP protocol request. Three complex types are defined and appear as parameters to these methods: *ControlPluginTransactionHandle*, which provides an interface to query the state of, and set results for, a transaction; *ControlPointType*, which is used to represent the state of a control point; and *ParameterType*, which is used to represent an experiment parameter; each method throws a *ControlPluginException* in case of error.

2.1 Methods defined in the Control Plugin Interface

The following methods are defined in the NTCP Control Plugin interface and must be implemented by the authors of a control plugin.

2.1.1 The propose method

```

public boolean propose(ControlPluginTransactionHandle transaction)
    throws ControlPluginException
  
```

When the NTCP server receives a *propose* (or *proposeAndExecute*) request, it validates the request, creates a new object to represent the proposed transaction and calls the plugin's *propose* method. The *propose* method should do any implementation-specific validation (such as verifying that all the control points that appear in the request actually exist, or that the control point parameters specified in the transaction request do not exceed any site-mandated maximum values). This method should return *true* if all site-specific conditions for accepting a proposed transaction are met, and *false* otherwise.

2.1.2 The execute method

```
public void execute(ControlPluginTransactionHandle transaction)
    throws ControlPluginException
```

When the NTCP server receives a request to *execute* a transaction (or when it successfully completes the proposal stage of a *proposeAndExecute* request), it validates the request, marks the transaction as *executing*, and calls the plugin's *execute* method, passing in the transaction's state.

The *execute* method should then begin to execute the transaction (asynchronously; for example, by starting a new thread for the execution) and return. If the execution completes successfully, the thread that is handling the execution should call the transaction's *setResultingControlPoints* method to set the transaction results; otherwise, it should call the transaction's *markTerminated* method to mark the execution as terminated without setting any transaction results.

Note: the plugin **must** call the transaction's *setResultingControlPoints* or *markTerminated* method when it has completed execution or has determined that execution has failed. The transaction will remain in the *executing* state, which will prevent the control points used in that transaction from being used in any subsequent transaction, until one of these two methods is called (or until the transaction's *rememberedUntil* timeout expires).

2.1.3 The cancel method

```
public void cancel(ControlPluginTransactionHandle transaction)
    throws ControlPluginException
```

When the NTCP server receives a request to cancel a transaction that is currently in the executing state, and the request's *interruptExecutingTransaction* flag is set, the server will call the plugin's *cancel* method to attempt to cancel the running transaction. If cancellation is not possible, the cancel method should throw a *ControlPluginException*.

2.1.4 The getControlPoint method

```
public java.util.Vector getControlPoint(java.util.Vector names)
    throws ControlPluginException
```

When the NTCP server receives a *getControlPoint* request, it calls the control plugin's *getControlPoint* method to query the actual values associated with that control point. The *names* argument is a Vector of control point names (of type String); the

`getControlPoint` method should return a *Vector* of *ControlPointType* objects, each of which contains the state of one of the requested control points.

2.1.5 The `setParameter` method

```
public void setParameter(java.util.Vector parameters)
    throws ControlPluginException
```

When the NTCP server receives a *setParameter* request, it caches the value of the parameter in the request and then calls the control plugin's *setParameter* method. The plugin may take any appropriate action – for example, it may forward the parameter on to a backend system, or throw an exception if the parameter name is recognized but the value is unexpected – or it may simply return without taking any action. The *parameters* argument is a *Vector* that contains a single *ParameterType* object containing the name and value of the parameter being set.

2.1.6 The `getParameter` method

```
public java.util.Vector getParameter(java.util.Vector names)
    throws ControlPluginException
```

When the NTCP server receives a *getParameter* request, it calls the control plugin's *getParameter* method to find the values of any requested parameters that are maintained by the plugin. The *names* argument is a *Vector* of parameter names, where each parameter name is a *String*. The control plugin should return a *Vector* of *ParameterType* objects, each of which contains a parameter name and value, or null if none of the requested parameters are known to the plugin. Note: if any of the requested parameters remain unresolved (that is, if there are any requested parameters that do not appear in the *Vector* returned by the call to the plugin's *getParameter* method), then the server will look in its own cache for the values of those parameters.

2.1.7 The `openSession` method

```
public void openSession(java.util.Vector parameters)
    throws ControlPluginException
```

When the NTCP server receives an *openSession* request, it calls the control plugin's *openSession* method, passing in a *Vector* of *ParameterType* objects containing the names and values of parameters for the new session. The *openSession* method may take any appropriate action (for example, it may initialize and pass parameters to a backend simulation).

2.1.8 The `closeSession` method

```
public void closeSession() throws ControlPluginException
```

When the NTCP server receives a *closeSession* request, it calls the control plugin's *closeSession* method, which may take any appropriate action.

2.2 Types Used by the Control Plugin Interface

The following object types are passed in as arguments to the methods in the control plugin interface.

2.2.1 ControlPointType

A *ControlPointType* object is used to specify values associated with a control point; these may be values representing an action requested on a control point, or measured/calculated values representing the state of a control point. A control point can be thought of as having a name and an array of (zero or more) values, each of which corresponds to (for example) a force or displacement along some axis. The methods within *ControlPointType* are described here.

```
public ControlPointType()
```

The *ControlPointType* constructor takes no arguments and creates an “empty” *ControlPointType* object (with no name or control points associated with it).

```
public void setControlPointName(java.lang.String controlPointName)
public java.lang.String getControlPointName()
```

The *setControlPointName* sets the control point's name; *getControlPointName* gets the control point's name (i.e., returns the name that was set by the most recent call to *setControlPointName*). Generally, *setControlPointName* will be called only once during the life of a *ControlPointType* object.

```
public void setControlPointType(ControlPointGeomParameterType[]
    controlPointType)
public void setControlPointType(int i, ControlPointGeomParameterType
    value)
```

The *setControlPointType* methods set the values associated with the control point (*ControlPointGeomParameterType* is described below). The first form sets the entire array; the second is used to set one value at a time.

```
public ControlPointGeomParameterType[] getControlPointType()
public ControlPointGeomParameterType getControlPointType(int i)
```

The *getControlPointType* methods get the values associated with the control point. The first form returns the entire array; the second returns the *ith* entry in the array.

2.2.2 ControlPointGeomParameterType

The *ControlPointGeomParameterType* object is used to represent a geometric parameter (such as “2 cm. displacement along the X axis”). The methods belonging to this type are described here:

```
public ControlPointGeomParameterType()
```

The constructor takes no arguments and creates an “empty” *ControlPointGeomParameterType* object.

```
public void setName(ControlPointParameterNameType name)
public ControlPointParameterNameType getName()
```

The *setName* method sets the name of the parameter (that is, the name describing what kind of parameter this object represents); *name* should be one of these statically-defined objects:

```
ControlPointParameterNameType.force
ControlPointParameterNameType.moment
ControlPointParameterNameType.displacement
ControlPointParameterNameType.rotation
```

The *getName* method returns the parameter’s name (the name set by *setName*).

```
public void setAxis(GeomAxisType axis)
public GeomAxisType getAxis()
```

The *setAxis* method sets the axis associated with this parameter; *axis* should be one of these three statically-defined objects:

```
GeomAxisType.x
GeomAxisType.y
GeomAxisType.z
```

The *getAxis* method returns the parameter’s axis (the axis set by *setAxis*).

```
public void setValue(java.lang.Float value)
public java.lang.Float getValue()
```

The *setValue* method sets the parameter’s value; *getValue* returns the parameter’s value.

2.2.3 ControlPluginTransactionHandle

A *ControlPluginTransactionHandle* object is used to represent the state of a transaction. A control plugin will never create a *ControlPluginTransactionHandle* object; however, control plugins do call methods on these objects.

```
public java.lang.String getName()
```

The *getName* method returns the transaction’s name. Some plugins may never call this method.

```
public java.util.Vector getRequestedControlPoints()
```

The *getRequestedControlPoints* method returns a *Vector* of *ControlPointType* objects (see above) representing the actions that were requested as part of this transaction.

```
public void setResultingControlPoints(java.util.Vector controlPoints)
    throws java.lang.Exception
```

The *setResultingControlPoints* method should be called when a transaction has finished executing successfully, to notify the server of the results of that transaction. The *controlPoints* argument should be a *Vector* of new *ControlPointType* objects representing the measured (or computed) values for that control point at the time that execution completed.

```
public void markTerminated() throws java.lang.Exception
```

The *markTerminated* method should be called when a transaction fails to execute successfully, to notify the server that the transaction has terminated unsuccessfully.

2.2.4 ParameterType

A *ParameterType* object is used to represent an experiment parameter; it can be thought of as a simple name-value pair. The methods of *ParameterType* are:

```
public ParameterType()
```

The constructor takes no arguments and creates an empty *ParameterType* object.

```
public void setName(java.lang.String name)
public java.lang.String getName()
```

The *setName* method sets the parameter's name; *getName* returns the parameter's name.

```
public void setValue(java.lang.String value)
public java.lang.String getValue()
```

The *setValue* method sets the parameter's value; *getValue* returns the parameter's value.

3 Control Plugin Use Cases

The following are some example servers that could be implemented using this plugin architecture.

3.1 A direct hardware control server

In this example, an NTCP server runs directly on a hardware control system.

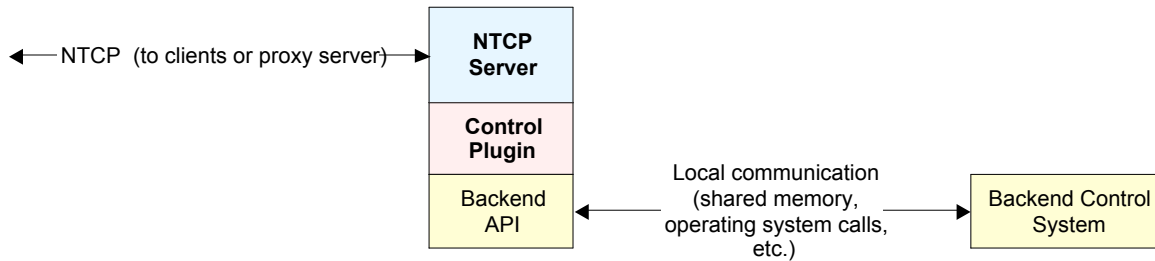


Figure 2: A direct hardware control server

In this case, the control plugin makes API calls to control a hardware control system directly.

Even if a site runs an NTCP server directly on a control system, that site may not wish to allow remote sites to access it directly (for security reasons, sites may choose to hide their control servers behind firewalls). In that case, the site may run a proxy server on their NEES-POP.

3.2 A Remote Hardware Control Server

In this example, the NTCP server runs on a system located on a different host than the backend control system and communicates with it over the network.

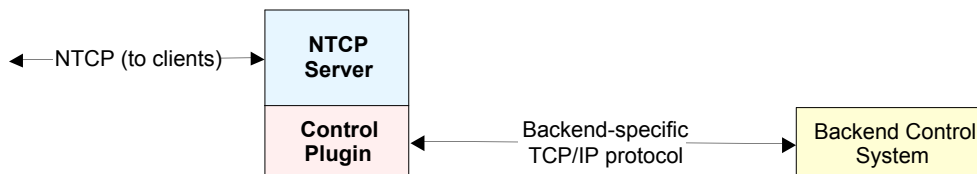


Figure 3: a remote hardware control server

In this case, the control plugin communicates with the backend control system using some TCP/IP protocol known to the backend system.

3.3 A Proxy Server

In this example, an NTCP server forwards requests to other NTCP servers. It may, in addition, enforce local policy (using the *policy plugin* interface, not described in this document) and translate control point names (e.g., from a distributed-simulation namespace into individual simulation namespaces).

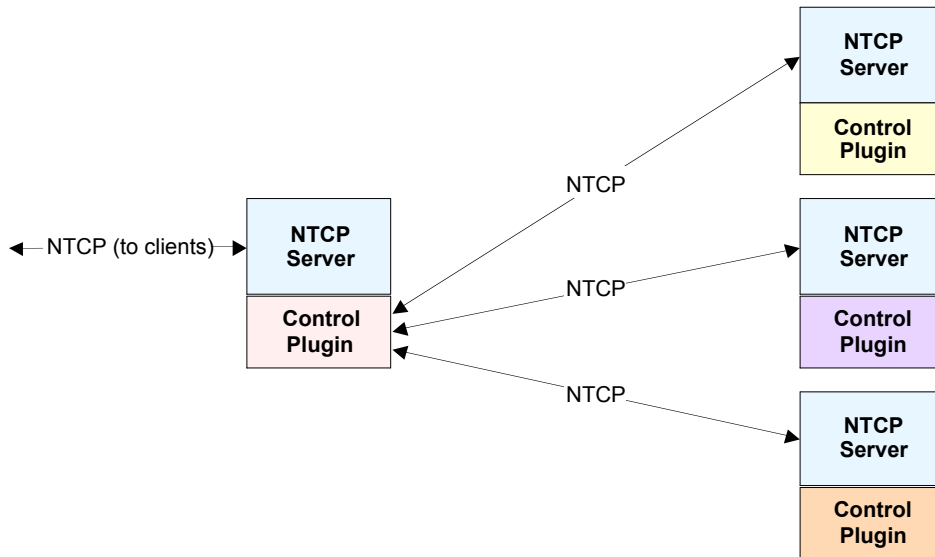


Figure 4: A Proxy Server

In this case, the control plugin for the proxy server forwards request to one or more other NTCP servers; each of these additional servers may be running a different control plugin.

3.4 A Computational Simulation

In this example, a computational simulation communicates using the NTCP protocol.

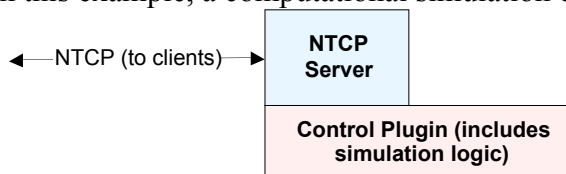


Figure 5: A computational Simulation communicating via NTCP

Writing individual simulations in this manner involves a certain amount of overhead; we suspect that researchers will prefer to use NTCP gateways into simulation-building tools.

3.5 An NTCP Gateway to a Simulation-Building Tool

In this example, the NTCP control plugin acts as “glue” connecting the NTCP server to a simulation-building tool. Experimenters can then use this tool to create simulations that communicate using NTCP.

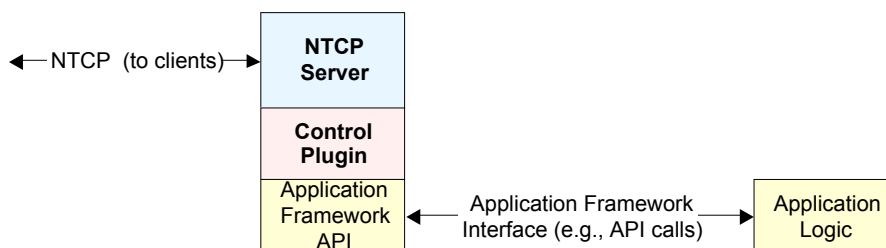


Figure 6: A gateway to an application-development framework

In this example, the local policy and control plugins call the APIs of an application development framework (the details are different for each framework). Experimenters can then build simulations by implementing their own application logic using that development framework.

3.6 Gateway to a Different Programming Language

The Control Plugin interfaced defined here is a Java interface. A “gateway” plugin would support the creation of plugins in a different programming language (such as C).

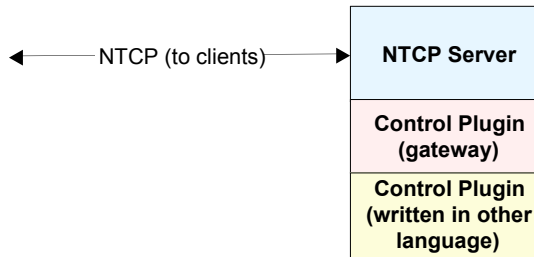


Figure 7: A language-gateway plugin

In this case, a new plugin API would be defined for a different programming language, and a “gateway” plugin would be written that made calls into that second API (for example, the Java *execute* method would call the equivalent function was equivalent in the C plugin API). Plugin implementers could then write plugins to the C API rather than writing Java.

Acknowledgments

We are grateful to Paul Hubbard, Erik Johnson, Benson Shing, and Bill Spencer for discussions leading to the development of this document.

¹L. Pearlman, M. D’Arcy, E. Johnson, P. Plaszczak, C. Kesselman. NEESgrid Teleoperation Control Protocol. NEESgrid Technical Report 2003-07. September, 2003.