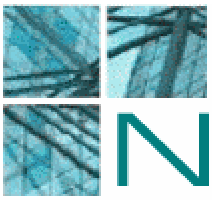

Technical Report NEESgrid-2003-08
www.neesgrid.org

(Whitepaper Version: 0.3
Last modified April 29, 2003)

A Plugin Interface for an NTCP Server

Laura Pearlman[1], Nabil Deeb[1], Carl Kesselman[1]

[1]USC Information Sciences Institute, Marina del Rey, CA

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.

Introduction

This document describes the overall software design of an NTCP server (as described in [the ntcp protocol document]), using plugins to implement local policy and to perform control operations. The goal is to create a generic NTCP server that, with appropriate plugins, can be used as:

- A server that controls hardware directly attached to the local system, 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).

The NTCP protocol consists of four requests: *propose*, used to initiate a transaction, *cancel*, used to cancel a transaction, *execute*, used to execute a transaction, and *transaction_status*, used to query the status and results of a transaction. Many parts of this service can be managed generically: enforcing generic policies (such as which users may send control requests for which resources), maintaining the state tables for transactions and resources, remembering transaction results and responding to queries, and mapping control point names to configuration information. However, there is some functionality server that cannot be implemented generically; we propose implementing this functionality using two kinds of plugin: a local policy plugin, to enforce local policy restrictions, and a control plugin, to handle the actual execution transactions.

1 Plugins

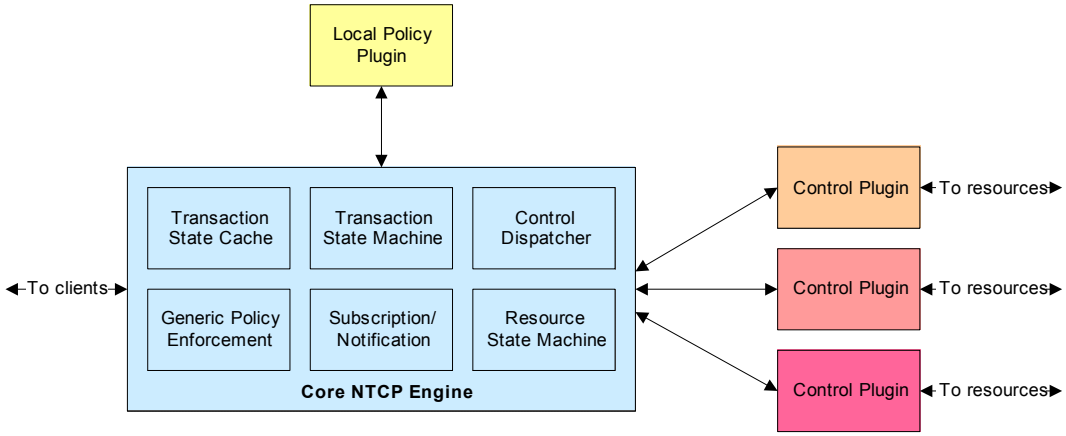

Figure 1: The NTCP server, with plugins.

An NTCP server must be configured to use a single local policy plugin; this plugin is called whenever a request is received, to enforce any local policy that cannot be expressed using the generic policy mechanism (an example of this kind of policy would be one that limited the amount of force that could be applied to a control point based on some computation).
An NTCP server must also be configured to use at least one control plugin. Control points are mapped to sets of resources; different resources may be mapped to different control plugins. When a transaction is executed (or an *executing* transaction is cancelled), the control plugin(s) associated with the resources involved in that transaction are called.

1.1 Local Policy Plugin Details

This plugin is called whenever any NTCP request is received and has passed all the server's generic authorization tests (if a request fails the generic authorization tests, then the server sends a failure reply to the client without proceeding further). This plugin is a java class with a constructor and a method called *check_local_authorization*.

When the NTCP server starts up, it calls the local policy plugin's constructor to create a local policy plugin object. The constructor takes one argument, a string containing configuration information (this string is parsed only by the plugin, not by the NTCP server, so the format of this string may be different for different local policy plugins).

Whenever the server receives any NTCP request, it calls the *check_local_authorization* method of the local policy plugin object, passing these arguments:

- A data structure representing the request and its arguments,
- A data structure representing the credentials used to authenticate the request, and
- A handle that can be passed to server utility functions to query control point states. [For July, no such utility functions will be implemented].

This function returns a data structure containing:

- A "yes or no" answer indicating whether the local policy authorizes the request, and
- A text string used for logging and for any status message returned to the client.

1.2 Control Plugin Details

A control plugin is a java class that includes a constructor and four methods: *initialize*, *execute*, and *cancel*.

When the NTCP server starts up, it calls the local policy plugin's constructor to create a control plugin object for each plugin configured into the server. The constructor takes one argument, a string containing configuration information. This string is parsed only by the plugin, not by the NTCP server, so the format of this string may be different for different control plugins; for example, one may expect the string to contain xml-encoded configuration information, while another may expect it to contain the name of a configuration file.

When the NTCP server receives a propose request for a transaction, it identifies the control points involved in the request, and the control plugin object associated with each of those control points. For each involved plugin, the server calls that plugin's *validate_propose_request* method, which takes these arguments:

- For each control point that is both involved in the current transaction and associated with this plugin, a data structure including the control point name and per-control-point arguments (that is, a virtual class should be defined for this, and the argument should be an object in a child of that class). [For July, this will be a simple structure that allows the specification of one or more of the following 12 quantities: Force, Moment, Displacement, and Rotation along the X, Y, and Z axes. The units used to express these quantities will be agreed on in advance].
- A handle that can be used to query the server's state and client authentication information.

This method will validate the propose request against any policy specific to this plugin (such as, "the result of applying this function to the parameters specified for control point X should be less than this value"). Note that more general policies, such as "User X may make requests involving control point Y", should be handled by the local policy plugin.

When the NTCP server receives an *execute* request for a transaction, it identifies the control points involved in the request, and the control plugin object associated with each of those control points. For each involved plugin, the server calls that plugin's *begin_execution* method, which takes these arguments:

- For each control point that is both involved in the current transaction and associated with this plugin, a data structure including the control point name and per-control-point arguments (the same data structure used by *validate_propose_request*).
- A handle that can be used to query the server's state and client authentication information.

The *begin_execution* method returns:

- The status ("success" – meaning execution has begun, or "failure", indicating that the arguments appear invalid), and
- If the status is "success", a (module-specific) data structure containing arguments to be passed to the plugin's *execute* method.

If the *begin_execution* request returned successfully, the server will then change the transaction's status to *executing* and call the plugin's *execute* method asynchronously, with three arguments:

- The name of the transaction (to be used by the *execute* method when reporting results),
- A *queue* object (a standard producer-consumer queue, with the *execute* method acting as the producer).
- The execution argument that was returned by the *begin_execution* method.
- A handle that can be used to query the server's state and client authentication information.

The *execute* method will:

- Do the actual work of executing the transaction (e.g., send a signal to a hardware controller, send a request to a remote server, or perform a computation),
- Create a data object for the results, consisting of the transaction name and structured results data for each involved control point. [For July, the per-control-point results data consists of 12 quantities: force, moment, displacement, and rotation along the three axes],
- Place the data object on the queue, and return.

If the NTCP server receives a *cancel* request for a transaction in the *executing* state, and the *cancel* requests indicates that the request should be cancelled even if it has begun executing, it will locate all the control points and control plugins involved in the transaction. For each plugin involved, the server will call that plugin's *cancel* method with a list of the control points that are involved in the transaction and associated with that plugin. The *cancel* method will attempt to cancel the request and return a status value.

When the NTCP server receives a *set_parameters* request, it will cache the parameter name and value and call the *set_parameters* method of each control plugin, with these arguments:

- The name of the parameter being set (from the NTCP request)
- A data object representing the parameter values (from the NTCP request)
- A handle that can be used to query the server's state and client authentication information.

What the *set_parameters* method does with this information depends on the implementation of each plugin; for example, it may save these values as static data effecting future calculations, or it may ignore them. The *set_parameters* method returns

a status – *success*, *unrecognized_parameter*, *unauthorized*, or *invalid_parameter*; however, it is permissible for a *set_parameters* method to simply ignore an unrecognized parameter and return *success* in that case. The status from *set_parameters* will be returned to the client as the status of the NTCP request.

When the NTCP server receives a *get_parameters* request for a parameter that is not cached, it will call the *get_parameters* method of each installed control plugin, passing the parameter name as an argument. The *get_parameters* method will return:

- A status indication: *success* or *notfound*
- A data structure containing the value of the parameter, if found.

If a plugin's *get_parameters* function returns the status *success,* then the value from that method will be returned to the client as part of the NTCP reply. [Open issue: if more than one installed plugin is keeping track of a named parameter, then the behavior isn't fully defined. For July, we will run configurations with only one plugin. For the future, we may enforce the limitation of only one plugin per server instance, or do something with parameter namespaces, or return an array of values].

2 Examples

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

2.1 A direct hardware control server

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

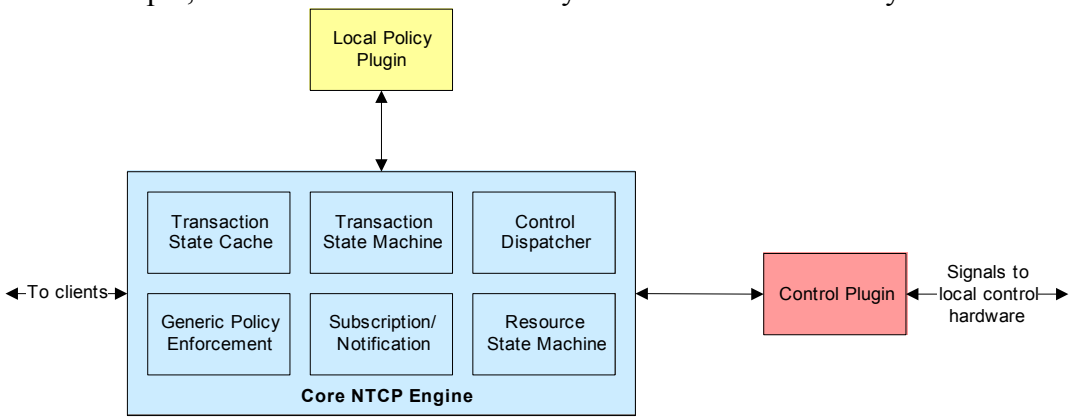

Figure 2: A direct hardware control server

The control plugin for this server is implemented as follows:

- The *initialize* function maps {control point, per-control-point-argument-type} pairs into {hardware slot number, parameters} pairs (e.g., "control point xyz, force along the X axis", to "the actuator at hardware slot 3" and additional parameters).

- The *begin_execution* method sanity-checks the results (e.g., it does not rejects requests that specify both force along the X axis and displacement along the X axis for the same control point).
- The *execute* method sends a signal to the appropriate board/slot, waits for results, and puts those results on the results queue.
- The *cancel* method either refuses to cancel executing requests, or sends a signal to the appropriate board/slot.

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.

2.2 A Proxy Server

In this example, an NTCP server forwards requests to other NTCP servers. It may, in addition, enforce local policy and translate control point names (e.g., from a distributed-simulation namespace into individual simulation namespaces). When connecting to backend servers, it may use its own credential (so that all policy is enforced by the proxy server; the backend servers can be configured to allow only the proxy server to perform control operations) or credentials delegated by clients.

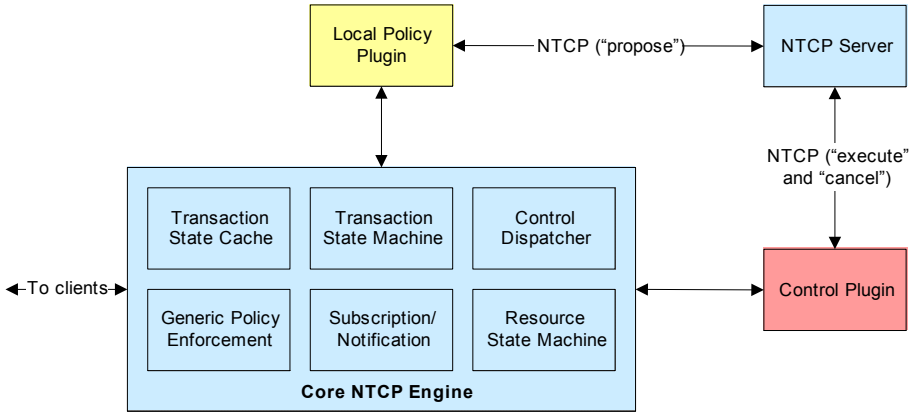

Figure 3: A Proxy Server

The local policy plugin for this server is implemented as follows:

- The *initialize* function creates a mapping of control point names into {remote server, remote-control-point} pairs.

- The *check local authorization* request:
  - Enforces local policy, and,
  - When checking a propose request, it translates the request into requests for the associated remote NTCP servers, sends the requests to those servers, and waits for the results. If the result of any of these results is failure, this function will cancel any requests that had been accepted as part of this transaction, and return an indication that the request is unauthorized.

The control plugin is implemented as follows:

- The *initialize* function maps control point names into {remote server, remote-control-point} pairs.
- The *begin_execution* method just returns success.
- The *execute* method sends execute requests to any remote servers involved in this transaction, waits for results, translates the control point names in the response, and puts the results on the results queue.
- The *cancel* method sends cancel requests to any remote servers involved in the transaction, then waits for results and puts them on the results queue.

[For July, we plan to implement a proxy server].

2.3 A Computational Simulation

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

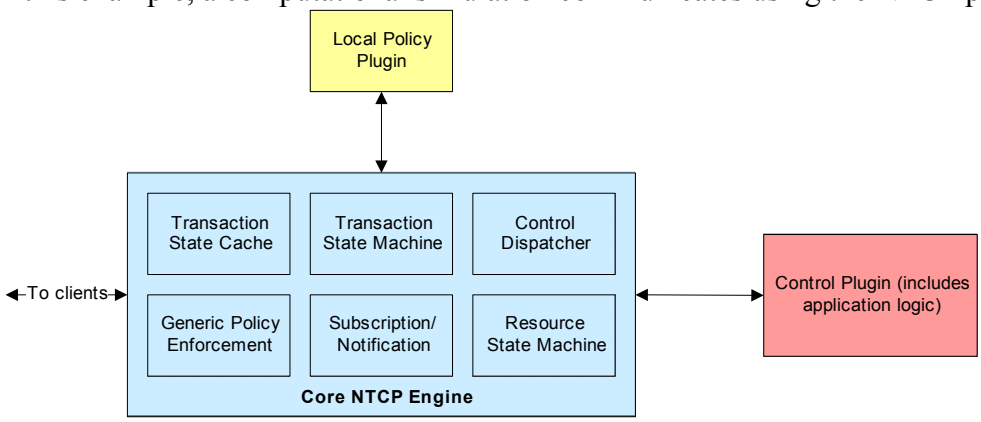

Figure 4: A computational Simulation communicating via NTCP

In this example, the control plugin is implemented as follows: the *execute* method performs the computations associated with each transaction, the cancel method either *fails* or rolls back any executing computations.

[For July, we do not plan to implement this kind of plugin].

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.

2.4 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.

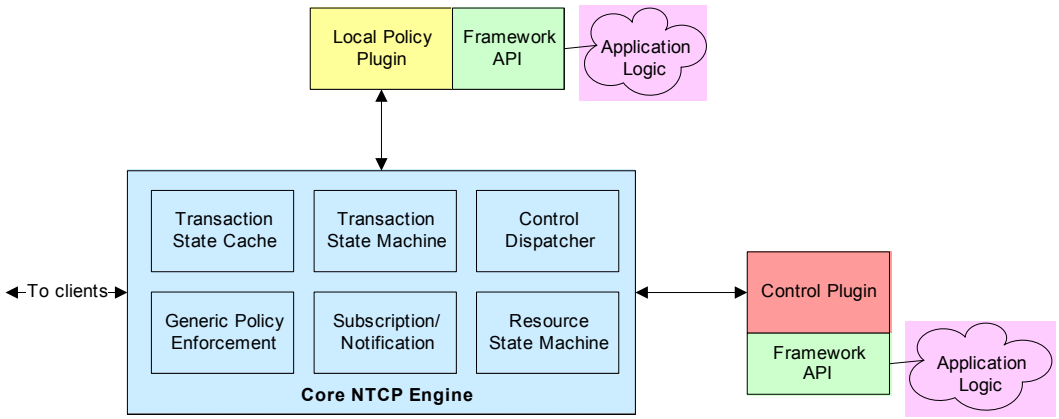

Figure 5: A gateway to an application-development framework

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.

[For July, we are planning to implement this kind of plugin, for Matlab].

2.5 Plugins Planned for July, 2003

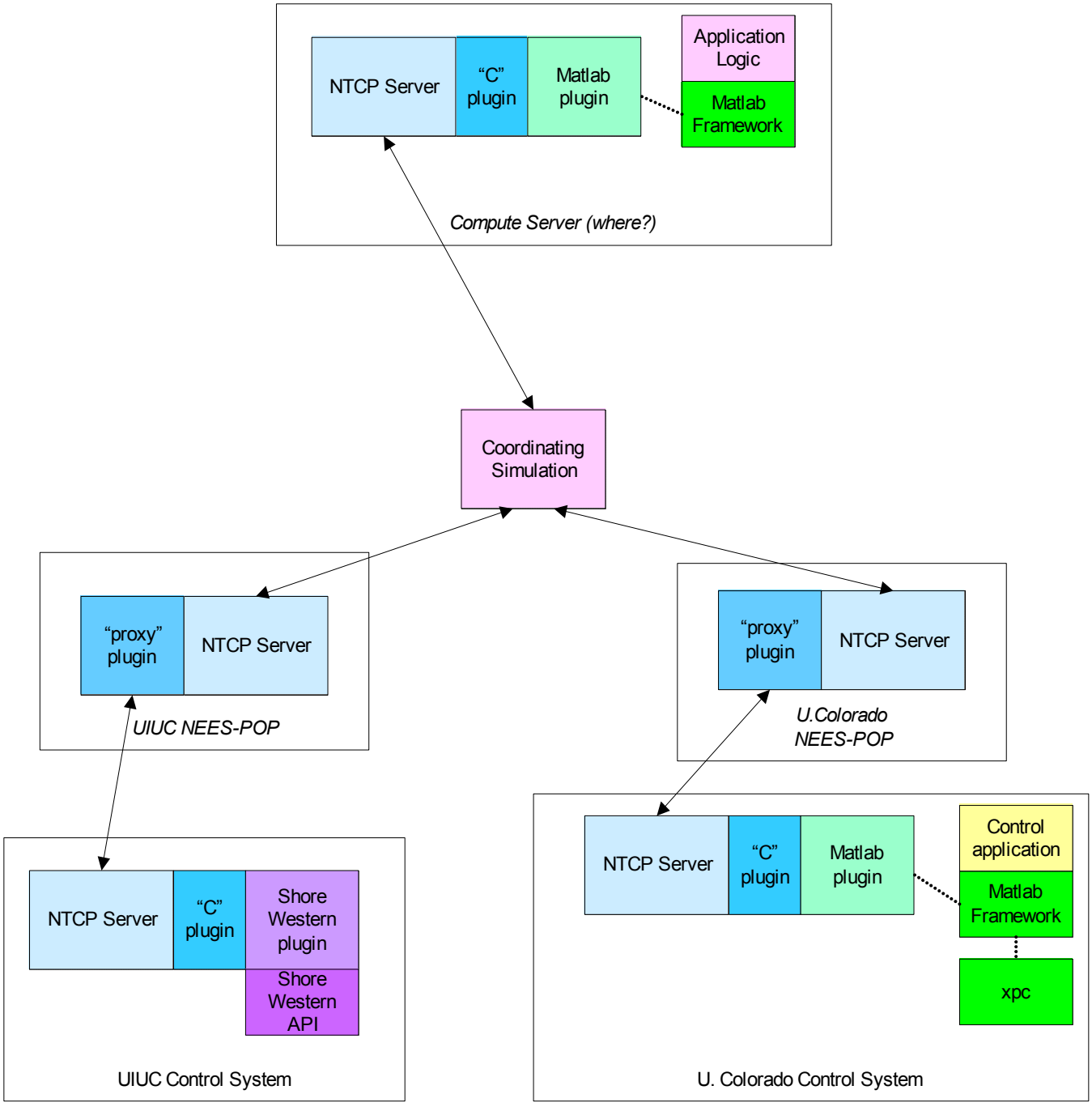

Figure 6: Plugins planned for July, 2003

For the July, 2003 experiment, we plan to have the following plugins:

- A "proxy" plugin, as described in Section 2.2, to run on the NEES-POPs at UIUC and the University of Colorado.

- A “C gateway control” plugin: a definition of the gateway plugin in the C language (implemented by having the server use jni calls).
- A “direct hardware control” plugin (as described in Section 2.1) to control Shore Western controllers at UIUC (using the C gateway control plugin).
- A local policy plugin and control plugin to act as a gateway to Matlab. This will be used both to implement computational simulations, and as part of the control mechanism at the University of Colorado. The Colorado control system will consist of an application using Matlab functions to receive NTCP requests and xpc to control their control hardware.

Acknowledgments

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