Building the National Virtual Collaboratory for Earthquake Engineering Research

NEESgrid

Technical Report NEESgrid-2003-07
www.neesgrid.org

(Whitepaper Version: 0.3)
Last modified: April 30, 2003)

NEES Teleoperation Control Protocol (NTCP)

Laura Pearlman[1], Erik Johnson[2], Carl Kesselman[1]

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

[2]University of Southern California, Los Angeles, CA

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.

1 Design Goals

The purpose of this protocol is to provide for control of simulations by remote applications, particularly in support of multi-site hybrid simulations.

A hybrid simulation may consist of a mix of computational simulations and physical simulations (experiments), and there are some cases in which it may be desirable to replace a physical simulation with a computational one (e.g., during a "dry run", or in the event that one site in a multi-site simulation experiences difficulties). For this reason, the same protocol should support both physical and computational simulations.

Different sites may choose different limitations on what operations are permitted during a physical simulation (e.g., maximum amounts of force to apply via an actuator), and it's generally not possible to "undo" an operation in a physical simulation. For this reason, the protocol should support negotiation of parameters at each site before any action is taken.

Communication over the network may be unreliable and may include delays. For this reason, the protocol should support closed-loop requests – that is, the protocol should support requests such as "move up 3 cm", as opposed to requests such as "move up at 10mph".

The protocol should not rely on the underlying transport to deliver messages reliably or in the same order in which they were sent (for example, the protocol should allow for recovery in the event that a TCP connection is dropped); the protocol should provide at-most-once semantics for its request.

2 Experiment Framework

A simulation design will include of a number of named virtual control points, on which actions can be specified. A virtual control point corresponds to a node in the simulation model; in a physical experiment, each control point will correspond to a location (on a specimen) on which forces can be applied by one more actuators. Within each computational or physical simulation, the virtual control point names must be unique. These names represent nodes in the simulation design; the same virtual control point names are used regardless of whether the actual simulation being run is a computational simulation or a physical experiment, or where the simulation is being run.

When (or before) a physical or computational simulation is run, a namespace will be assigned to that experiment instance (in NEESgrid, we expect that this assignment will eventually be handled via the NEESgrid metadata services). The control point names used in the NTCP protocol are names within these experiment instance namespaces. [For the July experiment, at least, we will use the convention that the "name" part of a control point name is the same as its virtual control point name].

Prior to the beginning of a distributed experiment, administrators at each local site running physical simulations will configure the NTCP server at that site to "understand" control point names, perform security-related configuration tasks (such as configuring who is allowed to send control requests for which experiments), and take any physical measures necessary for health and safety reasons. [For July, the only levels of authorization available will be "can send control requests"].

We expect that a distributed simulation will consist of a coordination module, which will coordinate requests for the distributed simulation as a whole, and one or more named computational or physical simulations as described above. The coordination module may itself be a single process or a distributed simulation, possibly utilizing a community scheduler.

We anticipate that the coordination module will pass through the following phases for each time-step of the simulation:

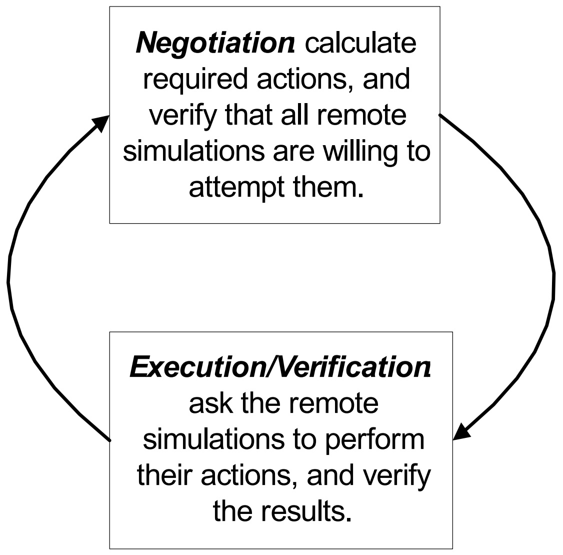
Negotiation calculate required actions, and verify that all remote simulations are willing to attempt them.
Execution/Verification ask the remote simulations to perform their actions, and verify the results.

Figure 1: Simulation phases during a time-step

In the *negotiation* phase, the coordination module negotiates with each remote site to determine the set of actions for the current time-step. The coordination module calculates the desired actions for each control point, and sends proposals to each NTCP server containing the parameters for the next actions of the control points controlled by that server. The server may accept the proposal, indicating a willingness to attempt the requested action, or reject the proposal, indicating an unwillingness to do so. The negotiation phase is successfully completed when all outstanding proposals have been accepted; at this point, the execution phase may begin.

In the *execution/verification* phase, the coordination module sends a request to each server to perform the actions agreed to during the negotiation phase. The servers attempt those actions and send status results back to the simulation, which may also gather additional data (e.g., sensor readings) to verify that the status sent back by each remote site is correct. When this phase is complete, the coordination module may begin the negotiation phase for the next time-step.

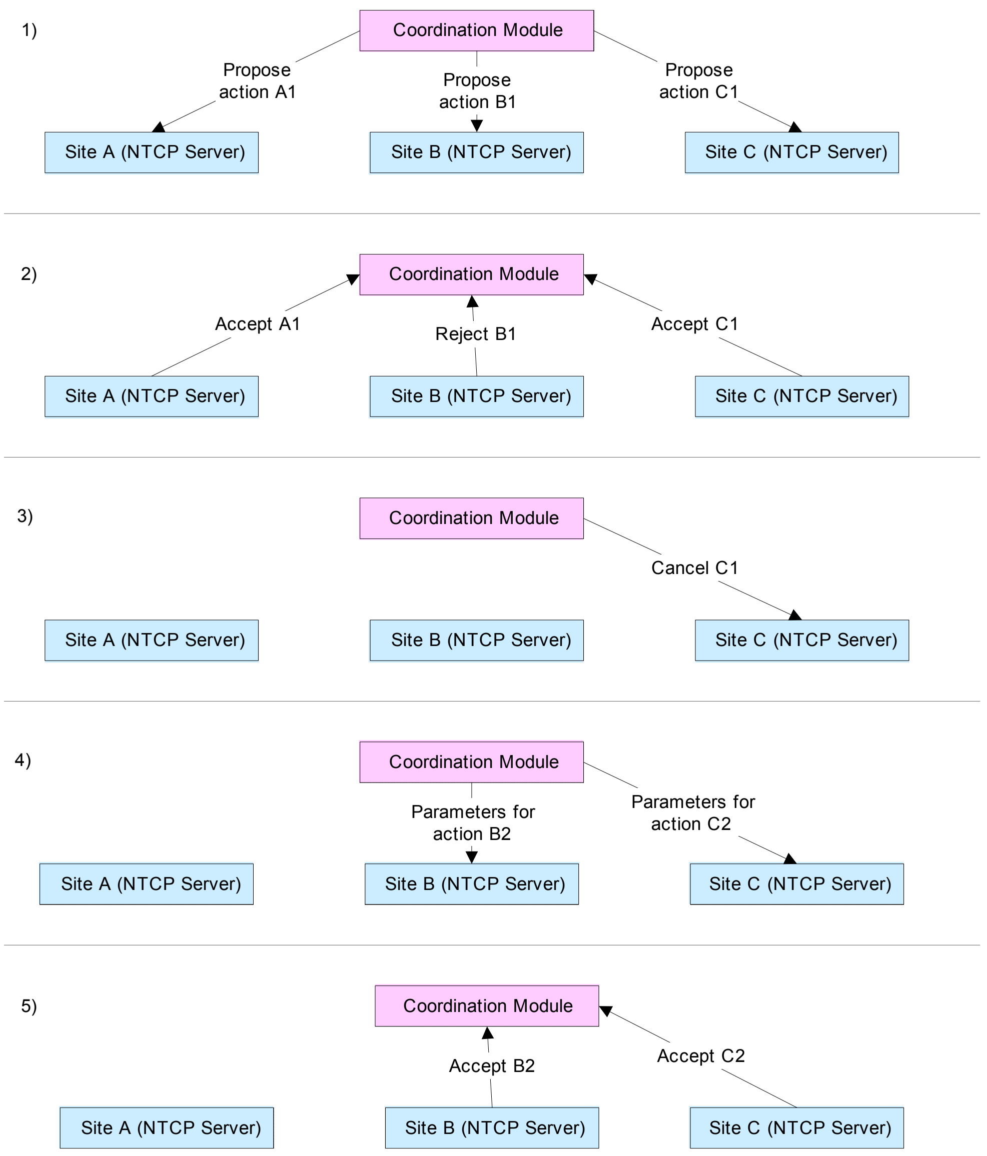

Figure 2: An example negotiation phase

Figure 2 shows an example negotiation phase. In step 1, the coordination module performs calculations to determine the desired actions at three remote sites for the current time-step, and sends the NTCP server at each site a message proposing the desired action for that site. In step 2, sites A and C each accept their proposals (A1 and C1, respectively), storing the parameters of the proposed action and replying with a message indicating their willingness to comply. Site B, however, rejects its proposal.

At this point, the negotiation phase has not completed successfully, and the simulation must deal with this failure, in some application-specific manner. The simulation might simply fail, or it might replace Site B with some numeric simulation for the remainder of the experiment. In this example, however, it calculates a new set of desired actions that is less demanding on Site B, but requires a change to the action requested of Site C. The simulation cancels the proposal previously accepted by Site C (step 3), and sends new proposals (B2 and C2) to sites B and C, respectively (step 4).

In step 5, sites B and C accept their new proposals. At this point, the simulation has an accepted proposal with each remote site (Site A has accepted A1, Site B has accepted B2, and site C has accepted C2), and the negotiation phase has completed.

2.1 The Execution/Verification Phase

Once the negotiation phase has completed, the coordination module sends a request to each remote site to execute that site's previously agreed-upon action. Each site's server attempts the action and reports status back to the coordination module. As the simulation receives status messages, it may gather additional information to verify that the status messages are correct.

3 NTCP Server State

The NTCP server maintains three kinds of state information: configuration state, experiment parameters, per-transaction state, and per-actuator state; this state information provides some serialization, discussed in section 3.5.

3.1 Configuration state

Configuration state includes the mapping of each *control point name* (the names that appear in NTCP protocol requests) to the (implementation-specific) information required to act on that control point. Configuration state also includes the mapping of each control point name to a set of *control point resources*; these are the resources that are involved in an action on that control point (e.g., in a physical experiment, the resources associated with a control point name may be a set of physical actuators). These mappings may be many-to-many: many control points may map to a single set of resources (e.g., a simulation may use two names for the same physical control point, or two simulations may share a set of actuators), and a single control point name may map to several sets of resources (e.g., a simulation may be mirrored within a distributed simulation).

Configuration state also includes policy information, such as maximum allowed request parameters and authorization information. [For July, the only policy information is who is allowed to perform control operations].

The management of configuration state is implementation-specific and is not addressed by the NTCP protocol.

3.2 Experiment Parameters

Closely related to configuration state are experiment parameters: parameters, not associated with any particular control point, that may effect a simulation. For example, in a computational simulation, the mass of a component may be an experiment parameter. Experiment parameters are cached by the NTCP server and are set and queried using the *get_parameters* and *set_parameters* requests.

3.3 Per-transaction state

A transaction can be in one of three possible states: *accepted*, *executing*, and *terminated*. In the most straightforward case, a client sends a *proposal* request to the server, which then accepts the proposal, sending the client an *accepted* reply and creating a new transaction in the *accepted* state. The client then sends the server an *execute* request for the transaction, which moves it to the *executing* state. When the execution completes, the server changes the transaction state to *terminated* and sends a reply to the client.

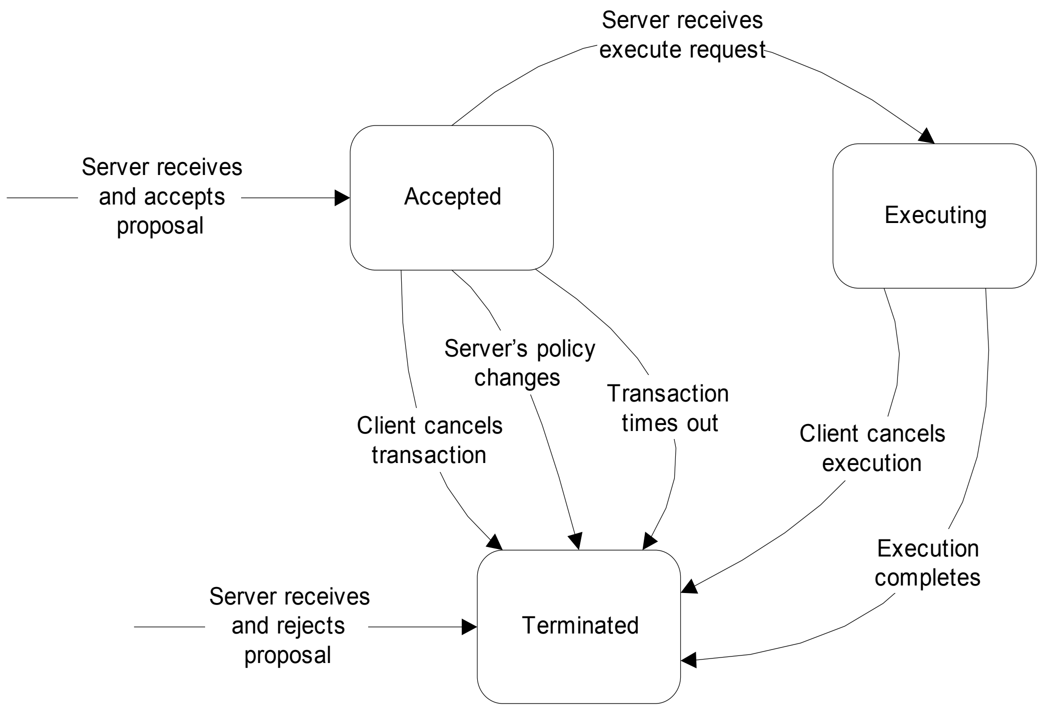
Server receives execute request
Server receives and accepts proposal
Accepted
Executing
Server's policy changes
Transaction times out
Client cancels transaction
Client cancels execution
Execution completes
Server receives and rejects proposal
Terminated

Figure 3: Transaction state transitions

Other state transitions are possible: timeouts or internal policy changes may move a transaction from the *accepted* state to the *terminated* state, and a *cancel* request may move a transaction from the *accepted* or *executing* state to the *terminated* state. An NTCP server may also receive a proposal and not create a transaction at all, if the proposal is syntactically incorrect, or if its transaction id is the same as the transaction id of an existing transaction.

3.4 Per-Resource State

The NTCP server associates *reservations* with control point resources; a reservation associates a resolved control point name with a transaction. Control point resource names were defined in section 3.1).
Reservations are exclusive: a resource may have at most one reservation at a time. A resource that has no reservations associated with it is said to be *available*.

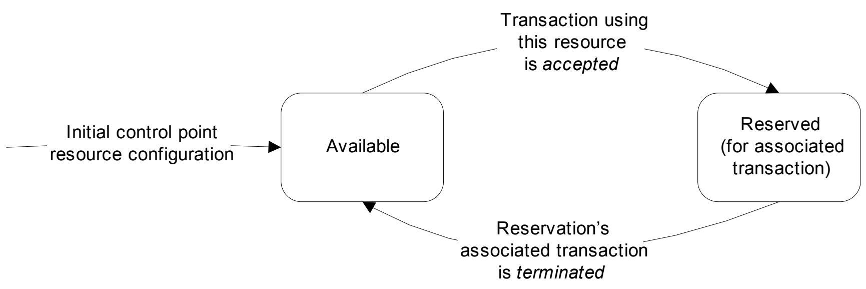
Transaction using this resource is *accepted*
Initial control point resource configuration
Available
Reserved (for associated transaction)
Reservation's associated transaction is *terminated*

Figure 4: Resource state transitions

Resource states and transaction states are closely related. Resource state depends on transaction state: a *reserved* resource becomes *available* when (and only when) the transaction associated with the reservation becomes *terminated*. Transaction state also depends on resource state: the NTCP server will not accept a proposal (that is, create a new *accepted* transaction) if it is unable to acquire reservations for the control point resources associated with all the control points involved in the transaction. The server must perform these state transitions atomically: at any point in time, if a transaction is accepted or executing, then all resources associated with that transaction must be reserved for that transaction, and if a resource is reserved, then the transaction associated with that reservation must be *accepted* or *executing*.

3.5 Serialization

This state model guarantees that:

- For each control point resource, at most one transaction involving that resource is executing at any time.
- Any two transactions that are executed on one control point resource are executed in the same order in which they were accepted.

This model does not make any guarantees about the order in which actions involved in a single transaction are executed (e.g., if a transaction involves control points A and B, then the action involving A may be executed before or after the action involving B, the two actions may be executed simultaneously, or the execution times may overlap).

This model also makes no guarantees about what happens in the time between two transactions involving the same client; for example, if a client sends requests creating and executing transactions A and then B involving control point X, there is no guarantee that some other client has not sent requests creating and executing transaction C involving control point X during the time after transaction A has completed and before transaction B has been accepted. If that kind of guarantee is desired, it could be provided via a community scheduler (see section 0).

4 The Protocol

The requests in this section comprise the NEES Control Protocol.

4.1 The Propose Request

The *propose* request is sent by the coordination module as part of the negotiation phase, to initiate a transaction and verify that a simulation is willing/able to attempt a specific action (or set of actions). The parameters passed with this request are:

- A new, unique transaction name.
- The proposal expiration time.
- The transaction expiration time (the time after which an *accepted* transaction becomes *terminated*).
- The per-control-point request parameters: a set of data structures, each of which contains a control point name and a typed structure that describes what kind of action is requested and lists the parameters for that action. [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].

There are three possible results of a *propose* request:

- A new transaction is created in the *accepted* state, with the transaction name, transaction expiration time, and per-control-point request parameters specified in the request. [Open issue: should we allow the server to change the transaction expiration time? For the July demo, the server won't change this.]
- No new transaction is created. This will occur if the transaction name corresponds to an existing transaction (even a terminated one).
- A new transaction is created in the *terminated* state with the parameters specified above. This will occur if the proposal is not compatible with the server's local policies, if the server receives the proposal after the proposal expiration time, or if the server is unable to acquire reservations for all the physical resources involved in the transaction, or if there is some other problem with the request (e.g., if the server doesn't understand the per-control-point request parameters).

The reply sent by the server will consist of the transaction name, an indication of the status (optionally including per-control-point status information [for July, no optional information will be returned]).

[Open issue: how to guarantee that the transaction names are unique. Does OGSA have a mechanism for this? Or should we have a separate request for the server to create a transaction name, or assign a namespace to the client?]

4.2 The execute request

The *execute* request is used to request that a server execute a transaction (that is, attempt the requested set of actions). The parameter passed with the *execute* request is:

- A transaction name: the name of a transaction in the *accepted* state.

There are two possible results of an *execute* request are:

- The named transaction is changed from the *accepted* state to the *executing* state, and the server begins executing the transaction.
- No transaction is executed or changes state. This occurs if the named transaction does not exist or is not in the *accepted* state, or if the client is not authorized to execute that transaction

The server does not wait for execution to complete before replying to the execute request; the results of the actual execution are communicated using subscription/notification.

4.3 The Cancel Request

The *cancel* request is used to cancel a transaction, i.e., to change that transaction's state to *terminated.* The parameters passed with the *cancel* request are:

- The name of the transaction to cancel.
- A flag indicating whether or not to cancel the transaction if it's in the *executing* state.

There are three possible results of a *cancel* request:

- The named transaction is changed from *accepted* to *terminated.*
- The named transaction is changed from *executing* to *terminated*, and the actual execution is interrupted; this occurs only if the flag argument indicates that the cancellation should occur even if the transaction is executing.
- No transaction changes state or is interrupted; this occurs if the transaction doesn't exist or is not *accepted* or *executing*, if the transaction is *executing* and the flag argument specifies that *executing* transactions should not be cancelled, if the transaction is *executing* and the server is incapable of interrupting it, or if the client is not authorized to cancel this transaction.

The reply from the cancel request includes status information.

4.4 The Set_parameters request

The *set_parameters* request is used to set experiment parameters. Its argument is a set of data structures, each containing:

- The parameter name (a string), and
- A typed data structure representing the value of the parameter (for July, this will simply be a string).

There are two possible results of a set_parameters request:

- The value of the parameter will be set, or
- The value of the parameter will not be set (because of a malformed request or an authorization failure)

4.5 Queries

The information from the following queries will also be available as service data elements.

4.5.1 Transaction_status

The *transaction_status* request is used to query the status of a transaction. The parameter passed with the *transaction_status* request is:

- The name of any existing transaction.

The reply includes the state of the transaction (*accepted*, *executing*, or *terminated*) and any additional available state related to that transaction (e.g., per-control-point results of a transaction that has executed successfully [For July, this will be a data structure containing the transaction name and transaction state, and, if the state is terminated, the termination status (*success*, *execution_failed*, or, for transactions that were cancelled or rejected, *never_executed*) and, for successful or failed transactions, the 12 measured values for force, moment, displacement, and rotation on each of the three axes].

4.5.2 Parameter_status

The *parameter_status* request is used to query the values of experiment parameters. The parameter passed with *parameter_status* is:

- Zero or more experiment parameter names.

The reply consists of a status value and a set of data structures of the same type used in *set_parameters*

- The parameter name (a string), and
- A typed data structure representing the value of the parameter (for July, this will simply be a string).

If any parameter names were specified in the call to *parameter_status*, then results for those parameters are returned; otherwise, a list of all parameters and their values is returned.

4.5.3 Control_point_status

The *control_point_status* request is used to query the current status of control points. The parameters passed with *control_point_status* are:

- A Boolean *immediate* flag
- Zero or more control point names.

The reply consists of a status value and a set of data structures consisting of:

- A control point name (a string), and
- A typed data structure representing the data for that control point (for July, the measured or calculated values for force, moment, displacement, and rotation on each of the three axes).

If any control point names are passed as input parameters, *control_point_status* returns the values for those control points; otherwise, *control_point_status* returns the values for all control points.

The server may keep a cache of control point values, and may use these values to reply to *control_point_status* requests; however, the *immediate* flag, if set, indicates that the client wishes the latest values, not cached values.

5 Security Considerations

The possible risks associated with a physical simulation are higher than the risks associated with most computing applications – accepting a "bad" request from a malicious (or simply broken) remote application could damage equipment or experiment systems, or in some cases even lead to serious injury. Although care will be taken to make the initial control service reasonably secure, they will be built using commonly-availably tools and will run on commodity operating systems, and cannot be guaranteed to be completely secure. Equipment sites should have appropriate non-software-based controls and procedures in place to safeguard their equipment and personnel.

6 Future Work: Transaction Queueing

Future versions of the NTCP service will support queues of transactions; the transactions in a queue will be executed in the order in which they appear in the queue. This will most likely be accomplished by extending the transaction state model to include an additional state (*queued_to_execute*) and creating a new request, *propose_queued_transaction*, with the same arguments as the *propose* request, plus one additional argument:

- The name of a *preceding transaction*.

The *preceding transaction* must be a transaction in either the *accepted, executing*, or *queued_to_execute* state and must be at the end of a queue (that is, it must not be the *preceding transaction* for any other transaction).

If the proposal is accepted, the newly-created transaction will be added to the queue following its *preceding_transaction* and will be in the *accepted* state. An *execute* request on this new transaction will result in its being placed in the *queued_to_execute* state; if the *preceding_transaction* of a queued_to_execute transaction terminates, the *queued_to_execute* transaction may begin execution as soon as all earlier transactions in its queue have terminated and all resources required for that transaction are available.

Appendix: Using a Community Scheduler with NTCP

Note: this section involves grid components that have not been implemented yet.

A community scheduler using the SNAP protocol[i] could optionally be used to provide serialization guarantees beyond those described in section 3.5 and to provide a more virtual model of actuators to the applications.

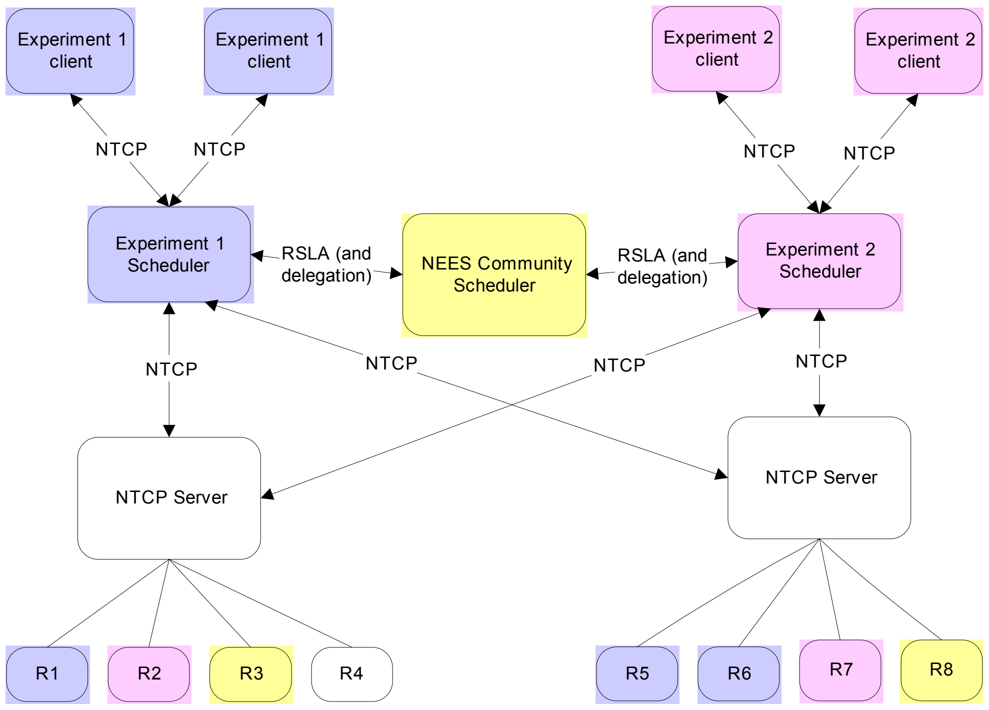

Figure 5: Using community schedulers with NTCP

The SNAP protocol uses Service Level Agreements (SLAs) to negotiate access to resources. A Resource Service Level Agreement (RSLA) is used to reserve a set of resources (such as cycles on particular compute servers or use of specific physical actuators); a Task Service Level Agreement is used to submit a task to be executed on an appropriate (but not explicitly specified) set of resources.

The NEES project could run a *community scheduler* to manage NEES resources: sites could configure their local servers so that some subset of their resources are controlled only by that scheduler. When a new experiment is to be run, an *experiment scheduler* could be created to handle control requests for that experiment. The experiment scheduler would negotiate RSLAs with the community scheduler (reserving specific control points for the exclusive use of that experiment). The RSLAs obtained by the experiment schedulers could be used to ensure that, for the duration of an experiment, the control points used by that experiment are used only by that experiment (this could be enforced by having each experiment scheduler run as a new, transient authentication identity, and having the community scheduler delegate authority to that identity). An experiment scheduler could then enforce any application-specific protocol used to coordinate actuator access among the components of a distributed application.

The client applications involved in an experiment would then communicate with the experiment scheduler using the NTCP protocol. The experiment scheduler would potentially perform two functions: maintaining its own, distributed-experiment-wide namespace for control point names, (facilitating the "hot-swapping" of one simulation for another, e.g., replacing a failed physical experiment with a computational simulation, without exposing the details to the client application), and enforcing any application-specific consistency scheme.

Acknowledgments

We are grateful to Nabil Deeb, Paul Hubbard, Benson Shing, and Bill Spencer for providing valuable insight and comments on drafts of this document.

[i] **SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems**. K. Czajkowski, I. Foster, C. Kesselman, V. Sander, S. Tuecke; *8th Workshop on Job Scheduling Strategies for Parallel Processing,* Edinburgh, Scotland, July, 2002.