26 Modeling Complex Application Behavior with Logic Scripts

You can model complex application logic and behavior with Python logic scripts that are embedded in Transaction Whiteboard model files.
Transaction Whiteboard includes an API of Python functions for modeling application behavior (such as sending messages or responding to timeouts) based on network, application, and user-specified inputs.

For more information, see the following topics:

- Logic Scripts: Example Use Cases
- Logic Scripts
- Troubleshooting Logic Scripts
- Scripting Framework
- Parameters
- Application Task Chaining
- Functions for Running Application Actions
- Functions for Accessing Information
- Transaction Whiteboard Scripting APIs
- Application_Base Class
- Connection Class
- Message Class
- Message_Stage Class
- Message_Status Class
- Node Class
- Resumable_Point Class
- Server_Job Class
- Stat Class
- Stat_Collect_Mode Class
- Stat_Type Class
- Modeling Web Services Dynamically

Logic Scripts: Example Use Cases

The following examples describe some of the capabilities of logic scripts:

- **Generate messages based on parameters or network/application conditions—**A tier can generate traffic based on user-specified parameters or in response to conditions in the network or the application.

  For example, you can create a parameter such as “number of pages” that determines the number of web pages requested. Before running QuickPredict or a simulation, a user sets this parameter to determine the number of pages downloaded during the task.

- **Generate traffic based on external data—**To model traffic based on existing network or application data, you can write Python scripts that read data from external sources and use the data to generate messages.

- **Application task chaining—**A tier can run another task based on Python code that specifies the conditions under which the child task is run. This enables you to model transactions in which tiers launch different child tasks in response to events in the application or the network. For example, you can create a login task that authenticates a login (specified using parameters). If the login succeeds, the task then runs a child task that models a web-based database query. Otherwise, the task ends.

- **Advanced delay modeling—**Every message has two attributes (processing delay and user delay) that specify the delay at the source tier before the tier sends that message. Because a logic script can modify these attributes, you can model tier delays in greater detail.

  For example, you can write logic scripts that model user behavior during a web-browsing task:

  — Web client sends request to web server

  — If web server sends a response, web client processes the response for 0.5 seconds (processing delay) and sends an acknowledgement

  — If there is no response after 20 seconds (user delay), web client sends a new request to a different server

- **“Go-to” and “looping” application execution—**You can write logic code to skip or repeat parts of an application task. This enables you to model behavior such as application-layer retransmissions:

  — Client sends request to server

  — If no response, client retransmits

  — Client tries two more times

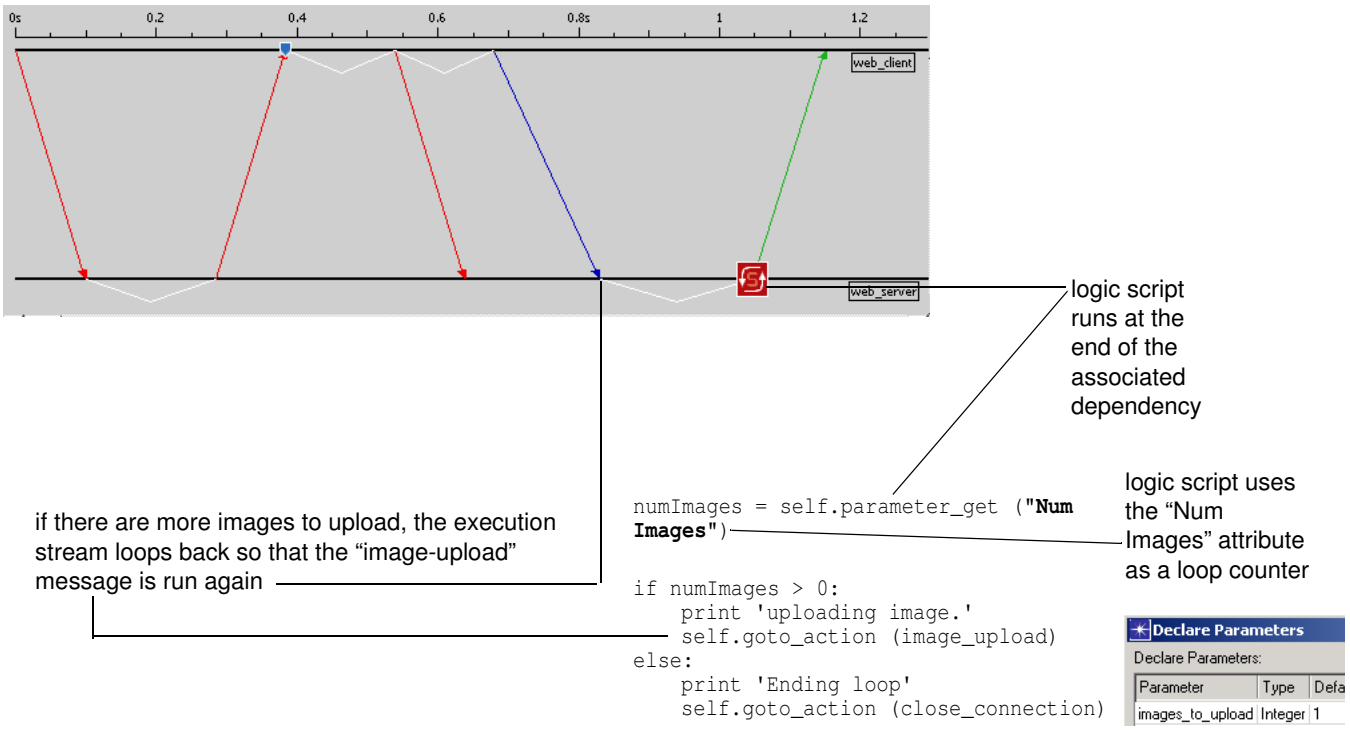
  — If it still gets no response, client skips the “response received” section

Logic Scripts

In the context of Transaction Whiteboard, a *logic script* specifies one or more actions and the internal logic that defines when those actions are run. Every logic script is associated with an application message dependency or message send. When a logic script is run, the actions defined in the script occur at the associated tier. Logic scripts on dependencies are executed either before or after the dependency. Logic scripts on messages are executed either before the message is sent or after the message is received.

The following figure shows a simple application that uploads images to a web server. This application has an application parameter called “Num Images.” Before running the application in QuickPredict or a discrete event simulation, a user can set this parameter to specify the number of images to upload when the application is run. The logic script then uses this attribute to repeat the image-file upload.

Figure 26-1 Logic Script Example



Creating a Logic Script

Every logic script is associated with a specific message or dependency. When the Transaction Whiteboard model is run in QuickPredict or a simulation, the logic script executes at the start or end of the associated message/dependency.

To create a new logic script, right-click on the associated message or dependency and choose “Add Logic Script at Start” or “Add Logic Script at End”. (You may need to zoom in to see the dependency clearly.)

Troubleshooting Logic Scripts

The Support Website maintains a set of frequently asked questions that contain information about troubleshooting, best practices and common usage for logic scripts.

To access the frequently asked questions, go to the Knowledge Base and search. To obtain a list of useful knowledge base entries, search for the string "Whiteboard"; you also might want to search based on a specific error message.

Scripting Framework

This section describes the general framework for writing logic scripts in Transaction Whiteboard.

- **Python Programming Language—**Logic Scripts are based on the Python programming language. Teaching the fundamentals of Python programming is outside the scope of this manual. However, Python is easy to learn. The Python Web site (http://www.python.org) is a good starting point. Some highlights of Python include the following:
  - — Python has built-in string, list, and dictionary support.
  - — Like C++, Python supports object classes with methods in addition to stand-alone functions.
  - — In Python, data is typed but variables are not statically typed.
- **Local and Global Variables—**When a variable is declared in a logic script, the variable is local to that script.

  To define global variables, open the Initialization Block (Scripting > Initialization Block). A variable declared in this block is global to all scripts in the Transaction Whiteboard model.
- **Helper Functions—**You can define helper functions that are run by other logic scripts. To do this, define the functions in the Function Block (Scripting > Function Block). Functions declared in this block can be called by any logic script in the Transaction Whiteboard model.
- **Global Definitions—**Use the Header Block (Scripting > Header Block) to import Python libraries and to specify global definitions.

Parameters

A parameter describes a task property. Parameters make a task configurable, because they can be set by a user before a QuickPredict run or a discrete event simulation. In a task-chaining scenario, a parent task can also set parameters for a child task before running it.

Unlike attributes for other types of objects, a parameter cannot be changed during QuickPredict or a simulation after the task is started. For an example of how parameters work, see Figure 26-1.

- **Declaring Parameters—**To declare new parameters, choose Scripting > Declare Parameters. You can specify a parameter name, type, default value, and description.
  - — Setting a Parameter—Unlike variables or external attributes, parameters cannot be changed after a task starts running. There are three ways that a parameter can be set before the task runs:
  - — Set the default value—Every parameter has a default value. You can specify the value for each parameter manually in the Declare Parameters dialog box (Scripting > Declare Parameters). The task will run with the default value unless it is changed by either of the following processes.
  - — Discrete event simulations—When you generate a scenario from a Transaction Whiteboard model, the scenario includes a Task Definition configuration object that lists the application parameters of the underlying task.

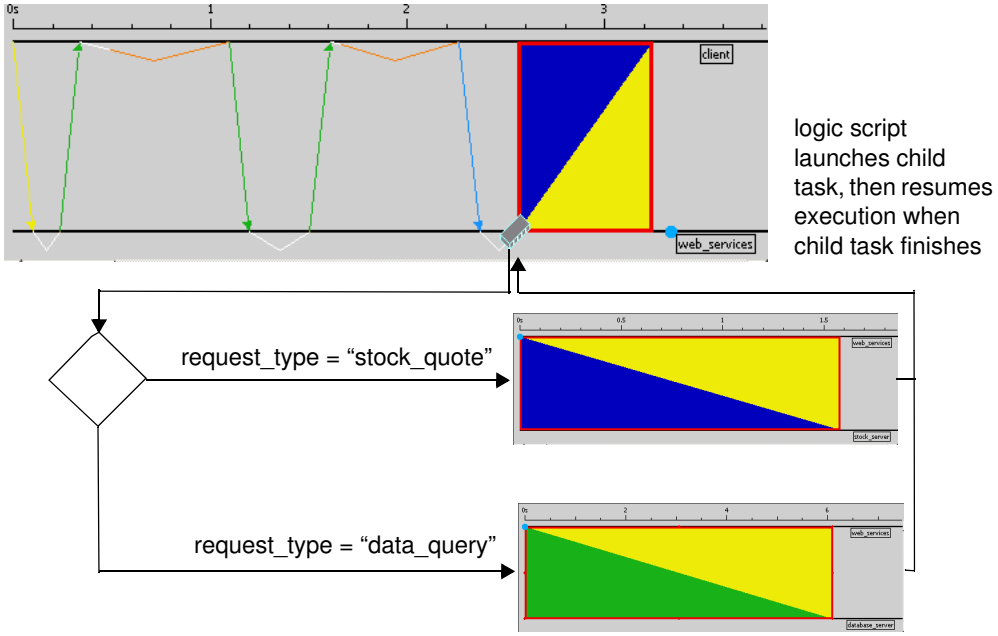
    To view and set the parameters related to a specific Transaction Whiteboard model:
    1. Right-click on the Tasks Definition configuration object and choose Edit Attributes.
    2. In the Attributes dialog box, search for the attribute.
- **Launching a child task—**The *invoke_child_task()* method includes a *task_params* argument, so a parent task can set application parameters in a child task before launching it.

Application Task Chaining

The Transaction Whiteboard API includes an *invoke_child_task()* method for launching tasks defined in external Transaction Analyzer or Transaction Whiteboard model files. You can use this method to create a “task chain” in which a parent task calls one or more child tasks. Task chaining is useful for modeling applications that span multiple tiers and consist of multiple tasks.

The following figure shows an example of task chaining. The parent task has an application parameter called request_type; depending on the value of this parameter, the logic script at the web_services (bottom) tier calls either of two child tasks. The result is a dynamic web-services task with three tiers, in which the parent task models the client-side traffic and the child task models the back-end traffic.

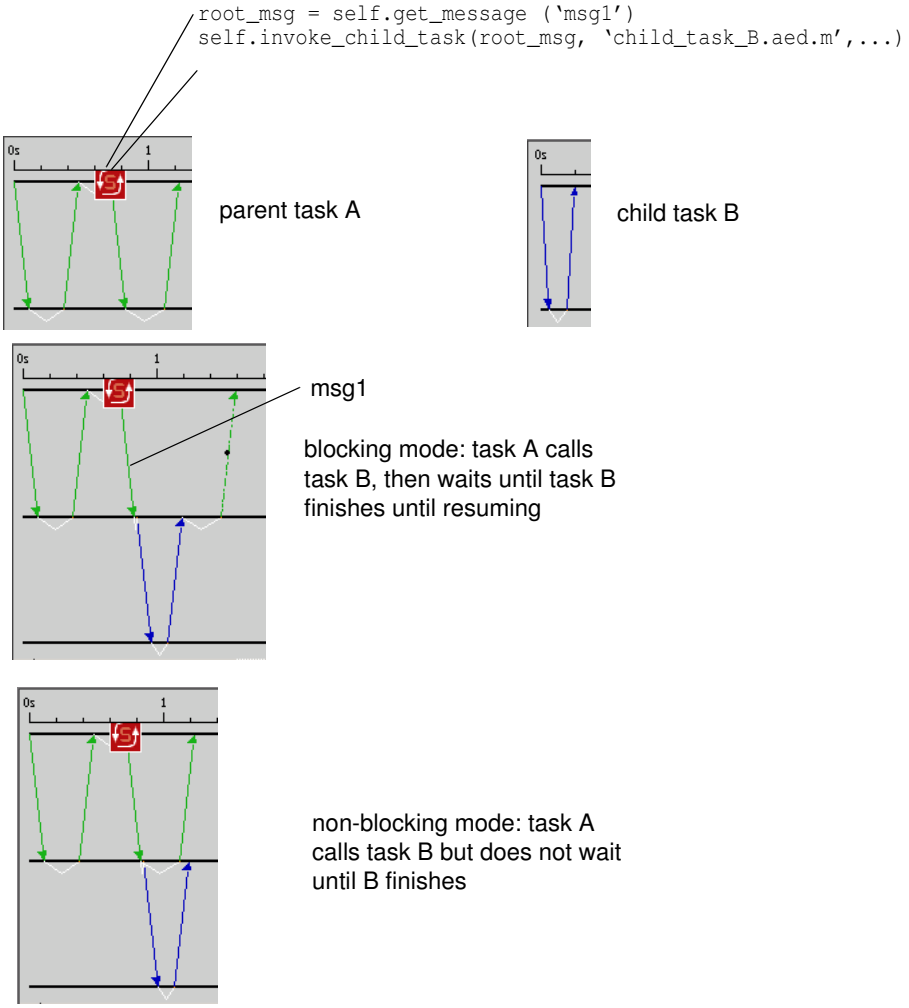
Figure 26-2 Task Chaining Example



Blocking and Non-Blocking Mode

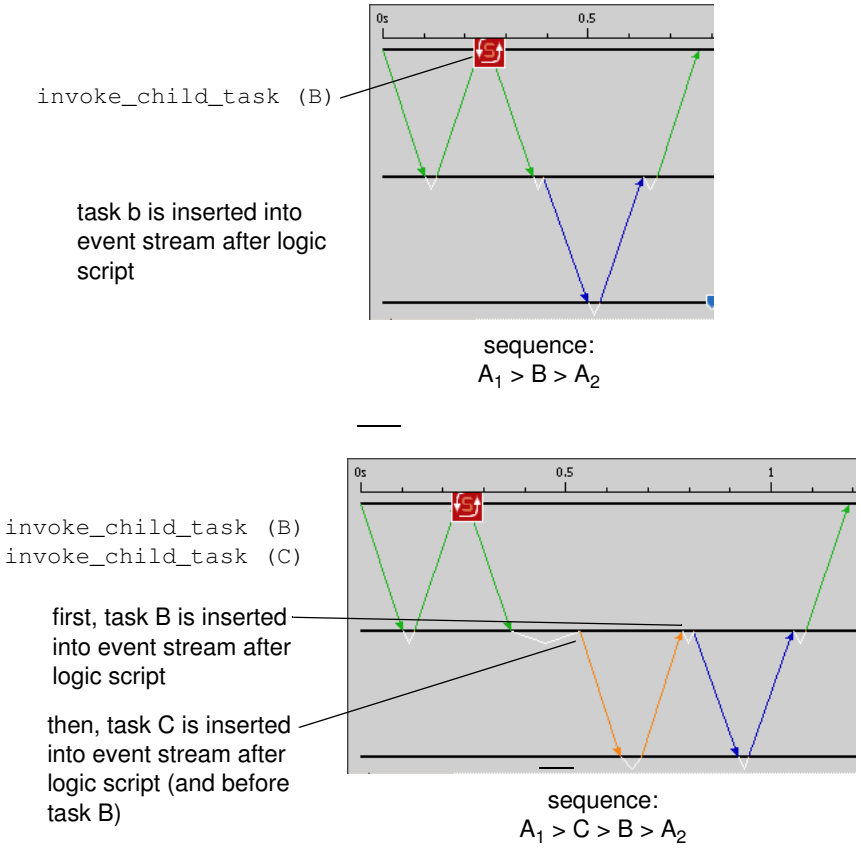
You can specify that a child task be launched in *blocking* or *non-blocking* mode. When a parent launches a child in blocking mode, execution of the parent task is blocked until the child task finishes. (The application shown in Figure 26-3 is an example of blocking mode.) When a parent launches a child in non-blocking mode, the parent resumes execution after launching the child and the two tasks run at the same time.

Figure 26-3 Effect of Launching a Child Task in Blocking and Non-Blocking Mode



The following figure shows the practical effect of launching a child task in blocking vs. non-blocking mode.

Figure 26-4 Launching Child Tasks in Blocking Mode in the Same Logic Script



Note—If a logic script calls multiple child tasks in blocking mode, the child tasks are inserted into the event stream in a stack-like (first-in-last-out) manner, as shown in Figure 26-4.

Final Response

The end of a child task can be difficult to determine in advance. To identify the end of a child task to its parent task, Transaction Whiteboard uses the concept of a *Final Response*. This is a designated message created by the child task (though not necessarily the last one sent). Receipt of the Final Response triggers notification to the parent task that the child task is complete (if such notification was requested by registering a completion callback function from the *invoke_child_task()* method). Note that the Final Response message has no effect on the child task, which might continue executing. The Final Response simply provides feedback to the parent task about the progress of the child task, allowing the parent task to resume execution.

Each child task has a designated Final Response message when Transaction Whiteboard saves the file. This message is chosen automatically by Transaction Whiteboard using simple network estimation to identify the last message to complete.

Logic scripts can set a different Final Response while a child task is executing. You can use the *set_final_response()* method to designate any message of the child task as the Final Response, or to clear the designation so that there is no Final Response.

Logic Scripts in QuickPredict vs. Discrete Event Simulations

You might find that the Transaction Whiteboard task produces different results when run in QuickPredict or a discrete event simulation. The following items summarize the differences in how QuickPredict and simulations model the events in the Transaction Whiteboard model:

- A discrete event simulation models the task behavior using a detailed network model specified in a scenario file. QuickPredict models the task behavior using its own simplified network model with user-specified measurements for bandwidth, latency, and other network characteristics. This difference has the following effects:
  - — If a logic function relies on a network-specific delay, a simulation models the delay based on the external network model. QuickPredict does not model network events and delays in detail. Therefore, logic functions that rely on network delays—for example, *register_timeout_callback()*—have no effect in QuickPredict.
  - — If a logic function relies on attributes or conditions in the network—for example, *get_attr()*—QuickPredict uses the default attribute value specified as an argument in the function call.
  - — When a Transaction Analyzer or Transaction Whiteboard model is run in a discrete event simulation, the application tiers correspond to nodes in the network model. If a logic script includes *set_attr()* calls that change node attributes, the network model will be changed. These calls have no effect in QuickPredict.
- QuickPredict does not model suspend/resume functionality defined using *suspend_child_actions()* and *resume_child_actions()*.
- Because QuickPredict does not access the simulation kernel, any calls to *dist_uniform()* return the mean value of the distribution.

Functions for Running Application Actions

The following table lists the types of actions you can define in a logic script, and provides references to relevant functions.

Table 26-1 Application Actions Supported by Logic Scripting

Action Type/Description	Related Functions
Send message	*create_message()*
Run or suspend a child task	*invoke_child_task()* *suspend_child_actions()* *resume_child_actions()*
Designate a child task's last message	*set_final_response()*
Go to a different action in the task	*goto_action()*
Respond to a timeout	*register_timeout_callback()*
Change a node or tier	*set_attr()* *set_tier_node()*
Task scheduling/execution	*schedule_function()*
Open/close a connection	*close()*
Output status information	*sim_message()*

Functions for Accessing Information

The following table lists the Transaction Whiteboard functions that you can use to access information from parameters, the current task (or a child task), and the network.

Table 26-2 Functions for Accessing and Publishing Information

Information Category	Description	Related Functions
Application parameter	Global parameter that can be configured by a user or a parent task	*get_parameter()*
Message	Message ID, start tier, or status	*get_message()*
Node/tier	Node/tier that sends or receives messages during a task	*get_nodes()* *get_tier_node()* *get_connections()* *get_state()*
Connection	Connection ID, status, and tag	*get_connections()*
Simulation status	Status information generated by the simulation kernel during a discrete event simulation	*sim_time()* *get_tier_names()*

Transaction Whiteboard Scripting APIs

The Transaction Whiteboard scripting application programming interface (API) is a set of classes and methods that support the creation of logic scripts in Transaction Whiteboard models. The API comprises a number of classes, methods, and properties that are used to:

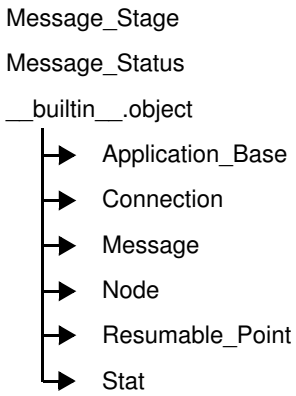
- Obtain information about an executing task
- Create or modify actions (such as messages and dependencies)
- Manage child tasks

The classes of the Transaction Whiteboard scripting API are contained in a module named Aps. Transaction Whiteboard automatically creates Aps objects as needed for logic scripts. You cannot directly instantiate any of the Aps classes.

Classes

The Aps module defines several classes used to create logic scripts. The following figure shows the class hierarchy.

Figure 26-5 Aps Module Class Hierarchy



The following table lists the classes and provides a link to a full description of each.

Table 26-3 Aps Module Class Summary

Class	Description
Application_Base Class	Base class for user-defined classes.
Connection Class	Provides access to and control of connections.
Server_Job Class	Provides the ability to create and process jobs.
Message Class	Provides access to Message properties.
Message_Stage Class	Enumerator class defining constants that represent where on a message an action is located.
Message_Status Class	Enumerator class defining constants that represent the status of a message.
Node Class	Provides access to network model nodes that are assigned to tiers.
Resumable_Point Class	Provides access to suspended child actions so that they can be resumed when needed.
Stat Class	Provides the ability to write custom vector statistics to the data files created during simulation.
Stat_Collect_Mode Class	Enumerator class defining constants that represent how statistics should be collected.
Stat_Type Class	Enumerator class defining constants that represent the scope of a statistic being registered

Note—Python does not have built-in support for enumerated data types. These are supported through classes that contain attributes only. This API has several such classes.

Keyword Arguments

Python allows the optional use of keyword arguments for methods. If a method supports keyword arguments, you can call it with only the arguments you need and leave out the unneeded ones.

To use keyword arguments, include the argument name, followed by an equal sign (“=”) and the argument value, in the method call. Keyword arguments can appear in any order. For example, the syntax of the *invoke_child_task()* method is:

```
self.invoke_child_task(parent_action, task_name, is_blocking, task_params, tier_map)
```

A call to this method using keyword arguments might look as follows. This call includes the three required arguments (in changed order), skips the fourth (optional) argument, and includes the last argument.

```
tiers = {'client':'web_client', 'server':'report_server'}
self.invoke_child_task (parent_action=action, is_blocking=True,
    task_name='child_task_2821.aed.m', tier_map=tiers)
```

Several methods in this API support keyword arguments, as noted in the *Syntax* section of such methods.

Application_Base Class

The Application_Base class is the base class for user-defined classes. When a task (`.aed.m`) file is loaded by a logic script, an instance of its class is created automatically. You cannot directly instantiate an Application_Base object.

Methods

- *create_job()*
- *create_message()*
- *dist_uniform()*
- *get_connections()*
- *get_message()*
- *get_nodes()*
- *get_nodes_compatible_with_tier()*
- *get_parameter()*
- *get_tier_names()*
- *get_tier_node()*
- *goto_action()*
- *invoke_child_task()*
- *sim_time()*
- *quit()*
- *register_receipt_callback()*
- *register_timeout_callback()*
- *schedule_function()*
- *set_final_response()*
- *set_tier_node()*
- *sim_message()*
- *suspend_child_actions()*

create_job()

Abstract Creates a server or mainframe job.

Class Application_Base

Syntax self.create_job(tier_name, job_name, total_exec_units)

Argument	Type	Description
tier_name	String	Corresponds to the tier name used in the Transaction Whiteboard model.
job_name	String	Corresponds to the job which must be configured in the simulated network.
total_exec_units	Integer	Specifies the total number of units that the job contains (optional).

Return Type Python Object Returns handle to the job.

Example

```
# Create a new job
if (req_type == 0):
job =self.create_job("db_server", "Database-SearchLastName")

elif (req_type == 1):
    job =self.create_job("db_server", "Database-SearchFirstName")

elif (req_type == 2):
    job =self.create_job("db_server", "Database-SearchSSN")

else
    print "Search Error: Unknown Search Criteria Encountered"
```

Details This method is used to create a server job or mainframe job from within Transaction Whiteboard. The arguments must correspond to the existing tier name and a job configured in the simulated network.

create_message()

Abstract Creates a message and schedules it to be sent.

Class Application_Base

Syntax self.create_message(size, src_tier, dest_tier, parent_action=None, parent_stage=Aps.Message_Stage.End, connection=–1)

Argument	Type	Description
size	Integer	Size of the message (in bytes).
src_tier	String	Name of the originating logical tier.
dest_tier	String	Name of the terminating logical tier.
parent_action	Action object	Action to designate as the parent of the message. Optional; defaults to the current action.
parent_stage	Integer	Stage in the parent action at which the message should be sent. Optional; defaults to Aps.Message_Stage.End.
connection	Connection object or Integer connection index	The connection the message should use. Optional; if not specified, the default connection for the tier pair is used.

This method supports keyword arguments.

Return Type Message The message that is created.

Example

```
new_msg = self.create_message(4096, "client", "server", action,
    Aps.Message_Stage.End)
new_msg.user_delay = 5.0
new_msg.tier_delay = 0.001
self.register_receipt_callback(new_msg, self.sim_message,
    ('Callback triggered','message receipt'))
```

Details This method creates a Message object and schedules it to be sent. If *parent_action* is the current action (the default), the message is sent when the logic script finishes. To make the message dependent on another message, set *parent_action* to the desired parent message. (That parent message should be one of the messages that was created in the given logic script).

In a callback function (such as a message receipt callback or a scheduled function) the parent action should be the “action” argument of the callback function. This makes an explicit connection between the parent action and the message being created.

The message that is returned has no dependencies (delays before the message is sent). You can use the Message Class properties *tier_delay* and *user_delay* to set dependencies for the message. You can also use the returned Message object to create children of the message or to register a receipt or timeout callback (with *register_receipt_callback()* or *register_timeout_callback()*, respectively).

dist_uniform()

Abstract Gets a random number.

Class Application_Base

Syntax self.dist_uniform()

Argument	Type	Description
Limit	double	Bound of the range of uniformly distributed value (optional)

Return Type double — Random number in the range [0.0, <*limit*>). This value is between 0.0 (inclusive) and the specified limit (exclusive).

Details If the limit argument is not supplied, this procedure assumes an upper bound of 1. If a negative value is specified for the limit argument, the returned value will be a uniformly distributed negative number and greater than limit.

get_connections()

Abstract Gets a list of current Connection objects.

Class Application_Base

Syntax self.get_connections()

Argument	Type	Description
		no arguments

Return Type PyList of Connection objects.

Details In a discrete event simulation, a connection will not appear in the list returned by this method until that connection has been used by a message. Connections also do not appear in the list after being closed (by the *Connection.close()* method).

In QuickPredict, all connections used by a task appear in the list at all times.

See Connection Class for more information about connections.

get_message()

Abstract Gets a message by tag.

Class Application_Base

Syntax self.get_message(msg_tag)

Argument	Type	Description
msg_tag	String	Tag assigned to the desired message in the Message Editor pane of Transaction Whiteboard.

Return Type Message Message with the specified tag.

Details This method raises a Python exception if it cannot find the specified message.

get_nodes()

Abstract Returns all nodes in a network model scenario that have specified attribute values.

Class Application_Base

Syntax self.get_nodes(criteria={})

Argument	Type	Description
criteria	Dictionary	Set of required attributes on which to filter the returned nodes. Optional; defaults to an empty dictionary.

Return Type List of nodes.

Example

```
criteria_dict = {'model':'ethernet_wkstn'}
node_list = self.get_nodes(criteria_dict)
self.sim_message('Number of nodes matching criteria: %d' % (len(node_list)))
```

Details Each element of *criteria* should specify an attribute name and the attribute value that returned nodes must have, in the form <name>:<value>.

This method raises a Python exception if *criteria* is not a Python dictionary.

get_nodes_compatible_with_tier()

Abstract Gets a list of nodes that can act as a specified tier.

Class Application_Base

Syntax self.get_nodes_compatible_with_tier(tier_name)

Argument	Type	Description
tier_name	String	Name of the tier-of-interest.

Return Type List of Node objects Nodes that can act as the specified tier.

Details This method returns a list of the nodes that can act as the specified tier (that is, nodes that have the Supported Services and AppTransaction Xpert Tier Configuration attributes set up for that kind of tier).

get_parameter()

Abstract Gets the value of a parameter of the current task.

Class Application_Base

Syntax self.get_parameter(name)

Argument	Type	Description
name	String	Name of the parameter to get.

Return Type Value of the specified parameter.

Details Task parameters are declared in Transaction Whiteboard with the Scripting > Declare Parameters operation. This method infers the data type of the parameter from the default value, so be sure to include a decimal when setting non-integer values.

This method raises a Python exception if the specified parameter does not exist.

get_tier_names()

Abstract Gets a list of all logical tier names.

Class Application_Base

Syntax self.get_tier_names()

Argument	Type	Description
		no arguments

Return Type List of Strings Tier names.

Details When called for a parent task, this method returns a list of all logical tier names in the task, such as “client”, “web_server”, or “app_server”. When called for a child task, only the tiers in the child task are included in the list.

get_tier_node()

Abstract Gets the node that corresponds to a tier name.

Class Application_Base

Syntax self.get_tier_node(tier_name)

Argument	Type	Description
tier_name	String	Name of the tier-of-interest.

Return Type Node object Node corresponding to *tier_name*.

Details This method raises a Python exception if no node can be found for the tier name.

goto_action()

Abstract Sets an action as the next thing to execute.

Class Application_Base

Syntax self.goto_action (action)

Argument	Type	Description
action	Action object or message tag (from the GUI)	The action to go to. This action must be in the same task (that is, in the same `.aed.m` file).

Return Type None.

Details This method can be used for looping or skipping over unwanted actions.

The *action* argument can be given in one of two forms:

- Action object—the action can be one of the following:
  - the current logic script (action)
  - a message (obtained by calling *action.get_message()* with a tag)
  - a child action of the previous message
- Message tag—a string tag assigned in the Message Editor pane of Transaction Whiteboard.

This method raises a Python exception if it cannot find the specified destination (*action*).

Note—If a logic script or a callback function (timer, message receipt, etc.) calls *goto_action()* in the same code block as *create_message()* or *invoke_child_task()*, the latter two function calls will be ignored. For example, if a logic script calls both *invoke_child_task()* and *goto_action()*, the child task will never be invoked. Similarly, if a feedback function calls both *create_message()* and *goto_action()*, the message(s) will never be created.

---

invoke_child_task()

Abstract Invokes a child task to the current action.

Class Application_Base

Syntax self.invoke_child_task(parent_action, task_name, is_blocking, task_params, tier_map, task_completion_function, task_completion_data)

Argument	Type	Description
parent_action	Action	Parent of the current action.
task_name	String	Name of the task to call. Should include the suffix (for example, `.aed.m`), but no path. If no suffix is given, `.atc.m` is assumed. The task file must be in a directory listed in the mod_dirs preference.
is_blocking	Boolean	Whether the child task should be blocking (True) or not (False).
task_params	Dictionary	Set of changes to parameters of the child task. Optional; defaults to an empty dictionary.
tier_map	Dictionary	Mapping of child task tiers to parent task tiers. Optional; defaults to an empty dictionary.
task_completion_function	bound method object	Python function to be called upon completion of the child task. Optional. For more information, see Completion Functions.
task_completion_data	tuple	Arguments to the callback function. Optional.

This method supports keyword arguments.

Return Type None.

Example

```
child_params = {'size':256, 'flavor':'vanilla'}
self.invoke_child_task(action, 'child_task_2821.aed.m', True, child_params, {},
self.my_completion_func, ("my_string",2))
```

Details This method invokes the child task at the end stage of the current message.

If *is_blocking* is True, the child task will be completely executed before the parent task continues. If *is_blocking* is False, the child task will be merged into the flow of events and executed concurrently with the parent task. (For a more detailed description, see Blocking.)

Use the optional *task_params* argument to specify values for the parameters of the child task. Any parameters not specified here will use the values specified in the *task_name* file. The elements of the dictionary should be given in <name>:<value> order.

Use the optional *tier_map* argument to specify a mapping of tiers in the child task to tiers in the parent. This method will override any mapping specified in Transaction Whiteboard with the Scripting > Manage Child Tasks operation. The elements of the dictionary should be given in <child_tier>:<parent_tier> order.

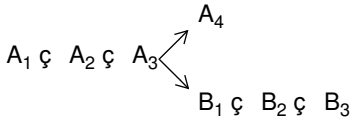
Use the optional *task_completion_function* and *task_completion_data* arguments to specify a callback function to be invoked when the child task is complete. (Completion is defined as receipt of the child task's Final Response message; see Final Response for details.) The *task_completion_function* must be defined in the Function Block.

This method raises a Python exception if it fails (usually because of a bad argument).

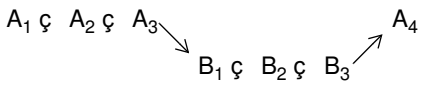
Blocking

Child tasks can be invoked in two modes, blocking and non-blocking. These modes determine how the actions in a child task will be merged with the actions of the parent task. For example, consider a task (A) comprising four actions (A_1, A_2, A_3, and A_4, of which A_3 is a logic script). Also consider a child task (B) with three actions (B_1, B_2, and B_3).

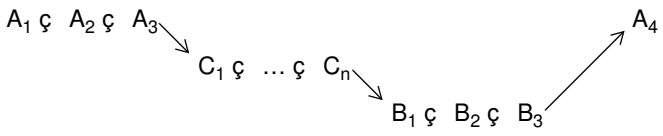
Non-blocking mode. If A_3 calls child task B in non-blocking mode, B_1 is appended to the child list of A_3 and executed at the same simulation time as A_4. The actions will be performed as shown:



Blocking mode. If, however, A_3 calls child task B in blocking mode, B is added to the action stack ahead of A_4. All of B's actions will be performed before task A continues:



If A_3 calls a second blocking child task (C) after calling B, C is added to the action stack ahead of B. Thus, even though it was called second, C's actions will be performed first:



Completion Functions

The *task_completion_function* must be defined in the Function Block, and have the following format:

```
def my_completion_func (self, action, data) :
    # The individual arguments can be accessed by unpacking the "data" tuple
    arg1, arg2 = data

    # At this point, arg1 is now equal to "my_string", and
    # arg2 is equal to 2.

    # Here would be the rest of the body of the function
```

sim_time()

Abstract Gets the current simulation time.

Class Application_Base

Syntax self.sim_time()

Argument	Type	Description
		no arguments

Return Type Double — Current simulation time, given as seconds from the begin simulation interrupt.

Details None.

quit()

Abstract Quits a task.

Class Application_Base

Syntax self.quit()

Argument	Type	Description
		no arguments

Return Type None.

Details This method schedules a quit action that will occur after the logic script is finished. A quit issued within a child task stops the parent task as well.

Note—Because of the way actions are scheduled, a quit might not happen immediately. For example, if you call *quit()* and then invoke a child task in blocking mode, the child task will be added to the action stack ahead of the quit and some or all of the child task's actions might execute before the quit. Therefore, it is better not to call anything after a *quit()*.

register_receipt_callback()

Abstract Registers a Python function to be called when a specific message is received.

Class Application_Base

Syntax self.register_receipt_callback(message, callback_function, callback_data)

Argument	Type	Description
message	Message object	Message whose receipt will trigger the callback.
callback_function	bound method object	Python function to be called.
callback_data	tuple	Arguments to the callback function. For more information, see Callback Functions.

This method supports keyword arguments.

Return Type None.

Example

```
new_msg = self.create_message(4096, "client", "server", action,
    Aps.Message_Stage.End)
self.register_receipt_callback(new_msg, self.sim_message,
    ('Callback triggered','message receipt'))
```

Details The Message object for *message* is obtained from the *create_message()* method.

If a timeout function also has been registered (with *register_timeout_callback()*) and the message is received after the timeout delay, both the receipt and timeout functions will be called.

Note—This method raises a Python exception if any arguments are invalid.If you are registering a callback for an explicit message, do not register that callback more than once. (In this context, an “explicit message” is a message created in the Transaction Whiteboard window and not dynamically created using *create_message()*.) Therefore, you should not register the callback function in a logic script that might be called more than once—for example, as part of a looping action implemented using *goto_action()*. A good place to register a callback function for explicit messages is the initialization block of the Transaction Whiteboard model file.

Callback Functions

The *callback_function* must be defined in the Function Block. If you make the following call:

```
self.register_timeout_callback(new_msg, self.callback_func, ("my_string",2), 10.0)
```

then the callback function format would be:

```
def callback_func (self, action, data) :
    # The individual arguments can be accessed by unpacking the "data" tuple
    arg1, arg2 = data

    # At this point, arg1 is now equal to "my_string", and
    # arg2 is equal to 2.

    # Here would be the rest of the body of the function
```

register_timeout_callback()

Abstract Registers a Python function to be called if a specific message is not received within a certain time.

Class Application_Base

Syntax self.register_timeout_callback(message, callback_function, callback_data, delta_time)

Argument	Type	Description
message	Aps.Message object	Message whose non-receipt will trigger the callback.
callback_function	bound method object	Python function to be called. For more information, see Callback Functions.
callback_data	tuple	Arguments to the callback function.
delta_time	Double	Length of time (in seconds) before the message is considered undeliverable.

This method supports keyword arguments.

Return Type None.

Details The function registered by this method will be called if *message* is not received within *delta_time* seconds from the time it is sent. If the message is received later, any function registered by *register_receipt_callback()* will also be called.

This method applies only to tasks used in discrete event simulations. In QuickPredict, it executes as a no-op and the timeout function is not registered.

Callback Functions The *callback_function* must be defined in the Function Block. If you make the following call:

```
self.register_timeout_callback(new_msg, self.callback_func, ("my_string",2), 10.0)
```

then the callback function format would be:

```
def callback_func (self, action, data) :
    # The individual arguments can be accessed by unpacking the "data" tuple
    arg1, arg2 = data

    # At this point, arg1 is now equal to "my_string", and
    # arg2 is equal to 2.

    # Here would be the rest of the body of the function
```

schedule_function()

Abstract Schedules a Python function for later execution.

Class Application_Base

Syntax self.schedule_function(delta_time, callback_function, callback_data)

Argument	Type	Description
delta_time	Double	Offset from the current simulation time (in seconds) at which the scheduled function should be called.
callback_function	bound method object	Name of Python member function to be called.
callback_data	tuple	Arguments to the callback function.

This method supports keyword arguments.

Return Type None.

Example

```
self.schedule_function(0.73, self.my_func, (1, 0.42, 'message receipt'))
```

Callback Functions The *callback_function* must be defined in the Function Block. If you make the following call:

```
self.schedule_function(0.73, self.my_func, (1, 0.42, 'message receipt'))
```

then the callback function format would be:

```
def my_func (self, action, data) :
    # The individual arguments can be accessed by unpacking the "data" tuple
    arg1, arg2, arg3 = data

    # At this point, arg1 is now equal to 1,
    # arg2 is equal to 0.42 and arg3 is equal to "message receipt"

    # Here would be the rest of the body of the function
```

set_final_response()

Abstract Sets a message as a child task's Final Response, or clears such a designation.

Class Application_Base

Syntax self.set_final_response(message)

Argument	Type	Description
message	Aps.Message object or string tag	Message to be set as the final response. No argument or "None" clears any existing final response designation.

Return Type None.

Details This method lets you set or clear the Final Response message for a child task. See Final Response for details.

The Message object for *message* is obtained from the *create_message()* method.

A child task will signal it completion only once. If a task contains a loop and sends its Final Response message several times, only the first instance of that message will be considered the Final Response. In such cases, you can delay the Final Response until the last repetition of the loop by clearing the Final Response prior to the loop and setting it again before the final iteration.

set_tier_node()

Abstract Sets a node as a logical tier.

Class Application_Base

Syntax self.set_tier_node(tier_name, node_name)

Argument	Type	Description
tier_name	String	Name of the tier for which to set the node.
node_name	String	Name of the node to set for the tier (see *Details*).

Return Type None.

Details You can set a tier node only before AppTransaction Xpert has resolved the node for that tier. Also, because setting a tier's node via Python causes the node to be resolved, you can set a tier node only once in a simulation. This restriction limits the use of *set_tier_node()* to two scenarios:

1) A Python function early in a task can redesignate a tier's node if the tier has not yet received or sent a message.

2) A parent task can set the node for a tier of a child task before calling that child task, as long as the tier is specific to the child task and not used in the parent task.

Note that a tier is supported by one or more network nodes in the network. *node_name* must be one of the following:

- The value of the "name" attribute of the node
- The source / client address of the node
- The hierarchical name of a node

You can use the get_nodes_compatible_with_tier() method to determine which nodes support a specific tier.

This method executes as a no-op in QuickPredict (because there are no nodes associated with tiers in this context). In a discrete event simulation, this method raises a Python exception if passed a name that does not exist, if the node is already resolved, or if the node has not been designated as being able to support a tier.

sim_message()

Abstract Prints a message on the standard output device.

Class Application_Base

Syntax self.sim_message(line0, line1=None)

Argument	Type	Description
line0	String	First line of message.
line1	String	Second line of message. Optional; defaults to None.

Return Type None.

Details None.

suspend_child_actions()

Abstract Suspends child dependencies of a designated stage of an action.

Class Application_Base

Syntax self.suspend_child_actions(action, stage)

Argument	Type	Description
action		Action whose child dependencies are to be suspended.
stage	Message_Stage	Which group of child actions to suspend. One of the enumerated constants of the Message_Stage Class.

Return Type Resumable_Point object — Object containing all suspended child dependencies.

Details This method prevents child actions of the specified stage from executing as they would in the normal course of events. Instead, they will wait until *resume_child_actions()* has been called. If the *resume_child_actions()* comes before the parent action is reached, there is no effect and there is the normal delay between parent and child. If the parent has already completed before *resume_child_actions()* is called, the delay time starts at the time of the resumption.

Suspending the children of an action that is already suspended will create another Resumable_Point for the same action. Regardless of how many Resumable_Points exist for an action, the children will be resumed with the first *resume_child_actions()* called. The other Resumable_Points will be treated as no-ops.

One situation that could cause multiple suspend calls would be repeatedly looping via *goto_action()*. If the suspended actions do not resume before *suspend_child_actions()* is called again, multiple Resumable_Points will be created. Thus, you should be careful about combining *suspend_child_actions()* with *goto_action()*.

One way of handling suspended traffic in a looping context is to create new messages with calls to *create_message()* and suspend them, thus ensuring that the suspended parent action is unique.

Connection Class

The Connection class is used to obtain and set information about connections in the task. It also provides a method for closing connections.

Methods

- *close()*

Properties

This class defines the following properties.

Table 26-4 Properties of Connection Class

Property	Type	Access	Description
id	Integer	read	Index of the connection in the list of open connections maintained by the task. The index of a specific connection can change as connections are opened and closed.
status	Boolean	read	Availability of the connection. Usually True, indicating an open connection. However, this can be False, indicating that the connection is being closed.

close()

Abstract Marks a connection as closed.

Class Connection

Syntax connection.close()

Argument	Type	Description
		no arguments

Return Type None.

Details This method executes as a no-op in QuickPredict.

Message Class

The Message class provides access to Message properties. This class treats the dependency (delay) before a message as part of the message, thus hiding the existence of dependencies (from the viewpoint of writing logic scripts).

Properties

This class defines the following properties.

Table 26-5 Properties of Message Class

Property	Type	Access	Description
connection	Integer	read	Index of the connection being used by the message.
end_tier	String	read	Name of the message's destination tier.
id	Integer	read	Message ID (as shown in the Transaction Whiteboard GUI).
size	Integer	read/write	Size of the message, in bytes. Must be >= 0.
start_tier	String	read	Name of the message's source tier.
status	Message_Status	read	Status of the message. One of the enumerated constants of the Message_Status Class.
tag	String	read	Name associated with a message in the Message Editor pane of Transaction Whiteboard.
take_over_children	boolean	read/write	If this property is true, message becomes the parent of the first post-script message. For more information, see *<message>.take_over_children Attribute*.
tier_delay	Double	read/write	Message delay caused by CPU processing, in seconds.
user_delay	Double	read/write	Message delay caused by user think time, in seconds.

<*message*>.take_over_children Attribute

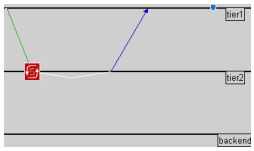
Message objects have a boolean attribute called take_over_children, which can only be modified by a logic script. This attribute is generally used to define the dependency relationship between the last message generated by a script and the first message after the script. If <*script_message*>.take_over_children = 0 (the default), there is no dependency relationship between the <*script_message*> and the first post-script message. If <*script_message*>.take_over_children = 1, then <*script_message*> becomes the parent of the first post-script message.

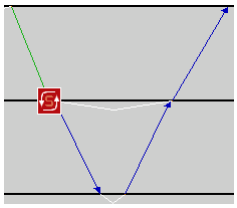
This attribute is useful when you want to model caching behavior at an intermediate node. For example, an HTTP cache server receives a request for a specific page:

- If the cache server has a copy of the page stored locally, it returns the page directly to the client. In this case, the request message (pre-script) is parent of the page-upload message (post-script) and <*last_script_message*>.take_over_children = 0 (since they are unrelated to the request/response messages).
- If the cache server has no copy of the request page, it retrieves the page from a back-end server (<*last_script_message*>) and returns it to the client (<*first_post_script_message*>). In this case, <*last_script_message*>.take_over_children = 1, because <*first_post_script_message*> depends on the completion of this message.

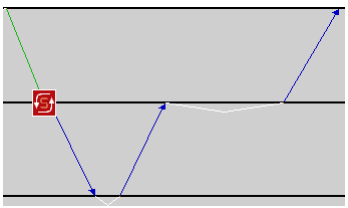
Figure 26-6 take_over_children: Example

```
actn1 = self.create_message (5000, "tier2", "backend", action)
actn2 = self.create_message (5000, "backend", "tier2", actn1)
```





actn2.take_over_children = 0
# actn2 is not parent of next message after script



actn2.take_over_children = 1
# actn2 is becomes parent of next message

Message_Stage Class

The Message_Stage class is an enumerator class. It defines symbolic constants that represent where on a message an action is located. Logic script actions always come at the end of a dependency.

Table 26-6 Message_Stage Constants

Variable	Type	Value
Start	int	0
End	int	1

Message_Status Class

The Message_Status class is an enumerator class that defines symbolic constants representing the different states of a message.

Table 26-7 Message_Status Constants

Variable	Type	Value
Unsent	int	0
Sent	int	1
Received	int	2
Unknown	int	3

Node Class

The Node class provides access to network model nodes that are assigned to tiers.

Methods

- *get_attr()*
- *get_state()*
- *set_attr()*
- *set_state()*
- *stat_register()*

get_attr()

Abstract Gets the value of an attribute of a network model node.

Class Node

Syntax node.get_attr(attr_name, default_value)

Argument	Type	Description
attr_name	String	Fully qualified name of the attribute being queried (see *Details*).
default_value	Double, Integer, or String	Value to be returned if the named attribute is not present in the Node object. The type of the default value should match the type of the node attribute.

Return Type Value of the attribute, if found; otherwise *default_value*.

Example

```
client_node = self.get_tier_node('client')
if client_node.get_attr('Client Address', 'Unknown') == 'new_york':
    client_node.set_attr('Client Address', 'big_apple')
```

Details *attr_name* must uniquely identify an attribute found on the node, including the full hierarchy for sub-attributes. If the value of *attr_name* is “name”, the hierarchical name of the node will be returned instead of the value of the attribute.

In QuickPredict, this method always returns *default_value* (because there are no nodes in this context).

get_state()

Abstract Gets the state associated with the specified key on that node.

Class Node

Syntax node.get_state(key)

Argument	Type	Description
key	String	Unique key associated wth this state.

Return Type (*user-specified*) The state associated with the specified node.

Details The returned state is accessible by all tasks. This method, together with *set_state()*, is useful for modeling job queues that support suspending and resuming child actions. A resumable_point can be added to the state of a node and later retrieved when it is time to resume the child actions.

If no state with the specified key is found, it returns None.

set_attr()

Abstract Sets the value of an attribute of a network model node.

Class Node

Syntax node.set_attr(attr_name, value)

Argument	Type	Description
attr_name	String	Fully qualified name of the attribute to set.
value	Double, Integer, or String	Value to be set on the named attribute.

Return Type None.

Example

```
client_node = self.get_tier_node('client', 'Unknown')
if client_node.get_attr('Client Address') == 'new_york':
    client_node.set_attr('Client Address', 'big_apple')
```

Details This method raises a Python exception if the data type of *value* does not match that of the specified attribute.

set_state()

Abstract Sets the state on the node using a unique key. Unique keys make it possible to set multiple states on the node.

Class Node

Syntax node.set_state(key, state)

Argument	Type	Description
key	String	Unique key associated with this state.
state	(user-specified)	State to set on the node.

Return Type None.

Details This method, together with *get_state()*, is useful for modeling job queues that support suspending and resuming child actions. A resumable_point can be added to the state of a node and later retrieved when it is time to resume the child actions.

If *key* corresponds to a state that is already set, *set_state()* overwrites that state with the current argument. To avoid a name collision, it is best practice to use a highly unique key. For example, instead of using *<key_string>*, use something like *<company_name>.<task_name>.<key_string>*.

stat_register()

Abstract Returns a handle that can be used to reference a local or global statistic.

Class Node

Syntax node.stat_register(stat_name, stat_type, collect_mode)

Argument	Type	Description
stat_name	String	Name of the statistic.
stat_type	Stat_Type	Type of statistic. One of the enumerated constants of the Stat_Type Class. Optional; defaults to Local.
collect_mode	Stat_Collect_Mode	Collection mode for the statistic. One of the enumerated constants of the Stat_Collect_Mode Class. Optional; defaults to All_Values.

Return Type Stat The statistic that was registered.

Example

```
# Get the client node as we will be writing this stat on the client
self.client_node = self.get_tier_node ('Client')
authentication_stat_handle =
    self.client_node.stat_register ('Authentication Period', Aps.Stat_Type.Local,
    Aps.Stat_Collect_Mode.Sample_Mean)
```

Details This function registers the given statistic as either a local or a global statistic. Local statistics are created for each node that corresponds to the tier running this application. Global statistics are cumulative and apply network-wide.

Choose the collection mode as follows:

Stat_Collect_Mode	Use for…
Sum_Over_Time	per-second value statistics
Sample_Mean	average value statistics
Time_Average	utilization statistics
Sum	summation statistics (e.g., “how many packets were sent?”)
All_Values	all other types of statistics

Resumable_Point Class

The Resumable_Point class provides access to suspended child actions so that they can be resumed when needed.

Methods

- *resume_child_actions()*

resume_child_actions()

Abstract Places the contained actions back into the chain of execution.

Class Resumable_Point

Syntax resumable_point.resume_child_actions()

Argument	Type	Description
		no arguments

Return Type None.

Details This method resumes processing of child actions that were suspended with *suspend_child_actions()*. Child actions that have been resumed begin with any defined dependencies. If child actions are suspended and then resumed before the actions would normally start, the suspension has no effect on the timing of child action processing.

Server_Job Class

This class provides the ability to create and trigger server and mainframe jobs from within the Transaction Whiteboard model. In addition, the *job.process()* method provides a way to create and process multithreaded jobs.

Methods

- *job.process()*
- *job.abort()*

job.process()

Abstract Processes a server or mainframe job.

Class Server_Job

Syntax job.process (callback_function, callback_data, exec_units)

Argument	Type	Description
callback_function	bound method object	Python function to be called after the job completes (usually defined in the function Block of the Transaction Whiteboard model)(optional).
callback_data	tuple	Arguments to the callback function. For more information, see Callback Functions.
exec_units	Integer	Specifies the number of units to execute out of the total number of units that the job contains (optional). Use this argument to process the job partially.

Return Type None

Example

```
# Set the start time for the job
start_time = self.sim_time ()

# Process the job as three parallel threads
args_thread1 = (start_time, 'thread1')
self.job.process (self.thread_completion_callback, args_thread1, 35)

args_thread2 = (start_time, 'thread2')
self.job.process (self.thread_completion_callback, args_thread2, 65)

args_thread3 = (start_time, 'thread3')
self.job.process (self.thread_completion_callback, args_thread3, 100)
```

Details Use this method to process a server or mainframe job. This method has four optional arguments. If no arguments are specified, the job executes to completion as a single thread.

When the processing completes, the *callback_func* Python function (if specified) is called and the *callback_data* arguments (if specified) are passed to it. For more information about callback functions, see Callback Functions.

Using the *exec_units* and *total_exec_units* arguments, you can split a job into multiple independent threads. For example, if a *job.process()* call specifies 15 *exec_units* and 60 *total_exec_units*, then 25% of the job begins executing as a single thread. A subsequent *job.process()* call with 30 *exec_units* (and 60 *total_exec_units*) executes another 50% of the job as another separate thread. To complete the job, 15 more *exec_units* must be specified in the remaining *job.process()* calls for a total of 60 units for the job.

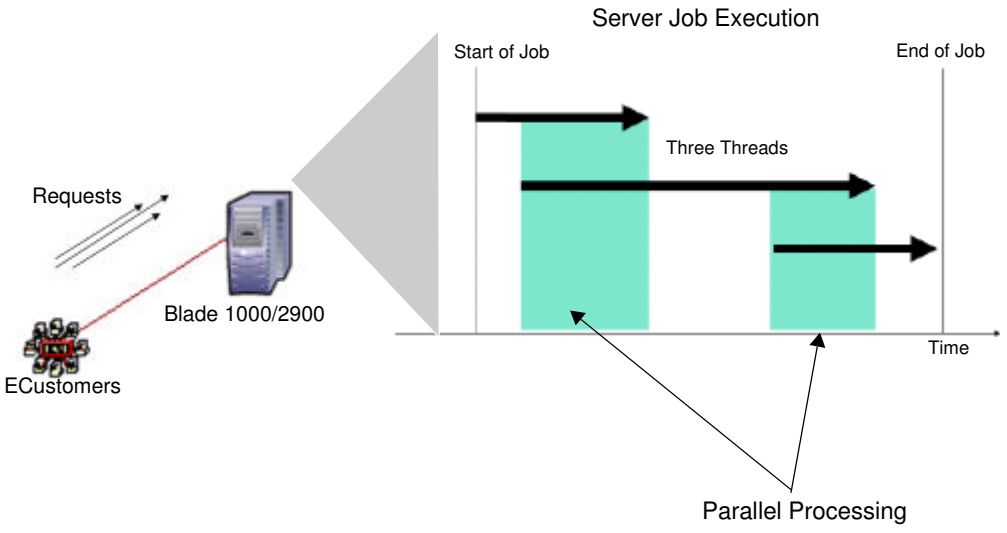
Note—When multiple calls are made for a job, the sum of the *exec_units* should never exceed the *total_exec_units* and the *total_exec_units* should remain constant throughout all calls for that job.

Parallel Threads

In this context, the term "thread" is defined as any part of a server or mainframe job that can run independently of the other part(s) of that job. This definition—although not strictly the same definition of a "thread" that a programmer may use (e.g., POSIX threads)—is particularly useful for calibrating server and mainframe jobs.

When two or more threads execute simultaneously, they are said to run in parallel. The following figure shows a representation of a server processing a multithreaded job. The shaded areas represent parallel processing during job execution.

Figure 26-7 Server Job with Parallel Processing



A job processed using parallel threads typically completes faster than the same job processed as a single thread. The most significant performance increase usually occurs when parallel threads execute on a machine with multiple CPUs.

Using Transaction Whiteboard and the Advanced Server or Mainframe Models, you can create and process multithreaded jobs.

The following sections describe two ways to create and process multithreaded server and mainframe jobs from within Transaction Whiteboard:

- Parallel Threads Using Scripting
- Parallel Threads Without Using Scripting

Parallel Threads Using Scripting

Using Python scripting, you can split a job into multiple threads. Using the *exec_units* and *total_exec_units* arguments in the *job.process()* method, you can create an arbitrary number of threads to process a job.

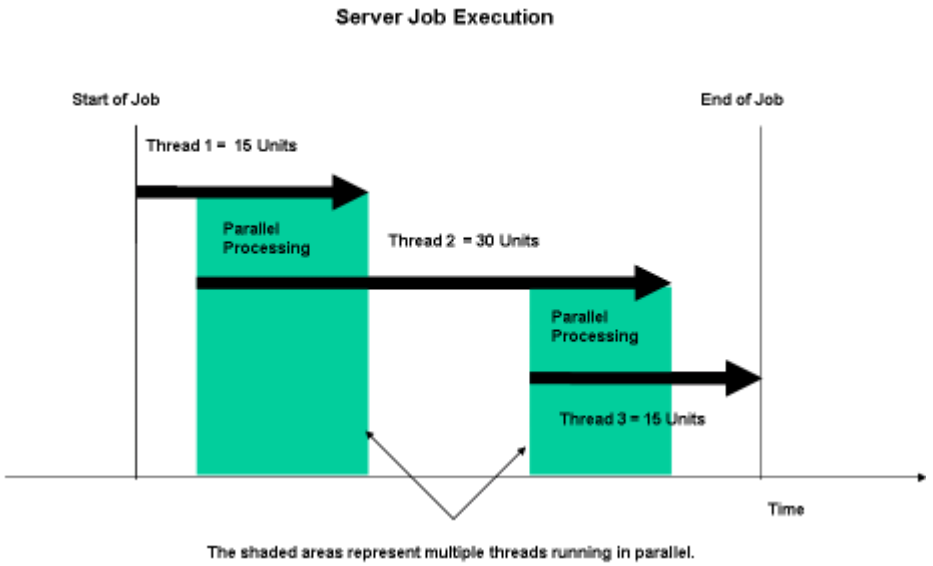
When a Python script calls *job.process()*, a thread is spawned. The fraction of the job executed by this thread is equal to the *exec_units* divided by the *total_exec_units*. If another *job.process()* call occurs before a previous *job.process()* call ends, the threads run in parallel. The job ends when all *job.process()* calls are executed to completion.

For example, the following three *job.process()* calls can specify individual threads for a particular job. The sum of the *exec_ units* must equal the *total_exec_units*. In this example 15 + 30 + 15 = 60. You can consider each thread as a partial completion of the total job.

```
job.process (callback_function,callback_ data,15,60) - Thread 1 (25%)
job.process (callback_function,callback_ data,30,60) - Thread 2 (50%)
job.process (callback_function,callback_ data,15,60) - Thread 3 (25%)
```

The following figure shows an example of how a job can execute in parallel threads.

Figure 26-8 Example Multithreaded Job with Parallel Processes Specified by a Python Script

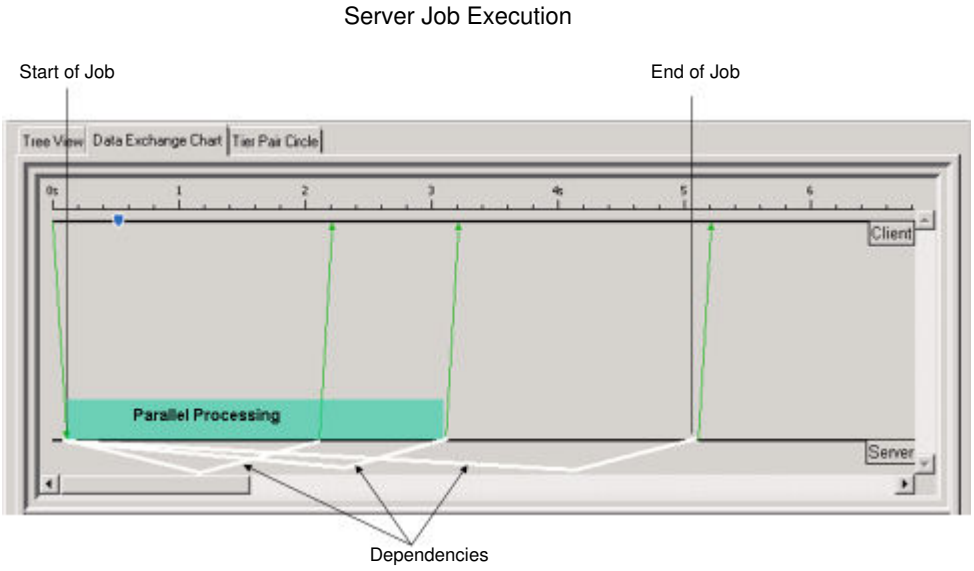


Parallel Threads Without Using Scripting

Using a GUI-based method in Transaction Whiteboard, you can create and process parallel threads without using Python scripting. This method can be easier to implement, however it is not as flexible as Python scripting. You cannot incorporate the same level of programmable logic as you can with Python scripting.

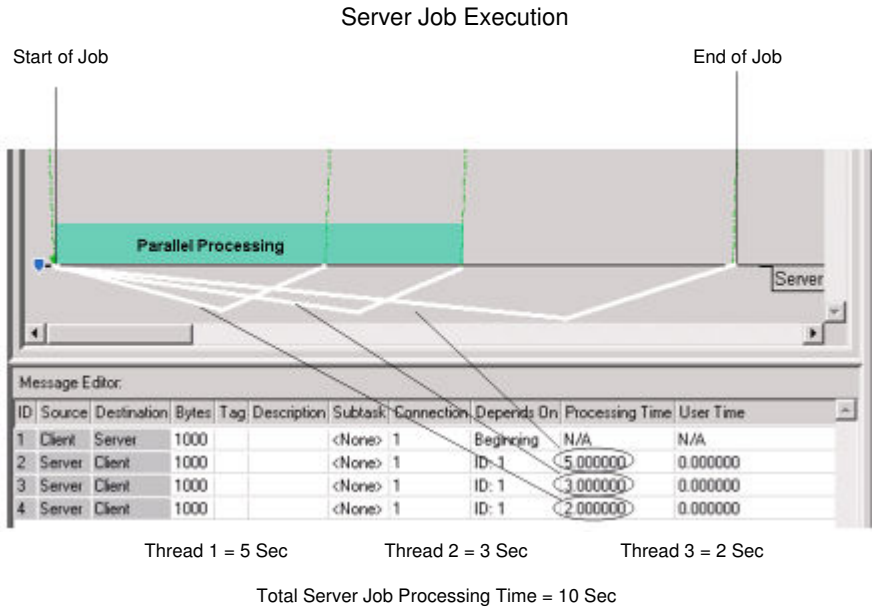
In Transaction Whiteboard, each dependency on the tier represents a thread. All the dependencies together represent the entire job. The following figure shows an example of a Transaction Whiteboard model that can execute the three threads in parallel.

Figure 26-9 Example Parallel Processes Specified by the Transaction Whiteboard Message Editor



You can use the Transaction Whiteboard Message Editor to specify the processing times for each job, as shown in the following figure.

Figure 26-10 Example Parallel Processes Specified by the Transaction Whiteboard Message Editor



For the example in the figure above, the job has three threads to process. Each thread makes up the given percentage of the total job:

```
ID 2 Processing Time 5 Secs / Total Processing Time 10 Secs
(5/10) - Thread 1 (50%)
ID 3 Processing Time 3 Secs / Total Processing Time 10 Secs
(3/10) - Thread 2 (30%)
ID 4 Processing Time 2 Secs / Total Processing Time 10 Secs
(2/10) - Thread 3 (20%)
```

Note that the processing times specified are not used to simulate the actual processing time for the job, but rather they are used to compute the relative percentages of the total job for each dependency. In other words, you can specify the processing times in any magnitude as long as the result is equivalent to the fraction of the total job you want to process for that thread.

The following list describes important considerations when creating and processing parallel threads using the GUI-based method:

1) This method of creating and processing parallel threads works only if there is no Python script in Transaction Whiteboard.

2) If there are Python scripts, the entire job is triggered as a single thread when the first dependency is encountered. This is intentional for the following reason—a Python script can dynamically generate new messages and dependencies, so that the pre-computation of the fractions for each dependency can no longer be valid.

3) If no server job is associated with Transaction Whiteboard through the server or mainframe attribute, then the absolute value of the processing times are used while simulating the dependencies.

job.abort()

Abstract Aborts the server or mainframe job.

Class Server_Job

Syntax job.abort ()

Argument	Type	Description
None		

Return Type None

Example

```
# If there is a timeout, abort the job
if (self.sim_time () - start_time > timeout):
    job.abort ()
```

Details Use this method to abort a server or mainframe job within Transaction Whiteboard.

Stat Class

The Stat (Statistics) class provides the ability to write custom vector statistics to the data files created during simulation.

Methods

- *write()*
- *write_t()*

write()

Abstract Writes a (time, value) pair to the specified statistic. The value is specified as an argument and the time is the current simulation time.

Class Stat

Syntax stat.write(value)

Argument	Type	Description
value	Double	Value to write to the specified statistic.

Return Type None.

Example

```
authentication_stat_handle.write (current_time - self.authentication_start_time)
```

Details This function appends a new (time, value) pair to an output vector for the specified statistic.

write_t()

Abstract Writes a (time, value) pair to the specified statistic. Both the value and time are specified as arguments.

Class Stat

Syntax stat.write_t(time, value)

Argument	Type	Description
value	Double	Value to write to the specified statistic.
time	Double	Value to be used as the abscissa of the output vector's new entry.

Return Type None.

Example

```
authentication_stat_handle.write_t
    ((current_time - self.authentication_start_time), task_completion_time)
```

Details This function appends a new (time, value) pair to an output vector for the specified statistic.

Stat_Collect_Mode Class

The Stat_Collect_Mode class is an enumerator class. It defines symbolic constants that represent how statistics should be collected.

Table 26-8 Stat_Collect_Mode Constants

Variable	Type	Value
All_Values	int	0
Sum_Over_Time	int	1
Sample_Mean	int	2
Time_Average	int	3
Sum	int	4

Stat_Type Class

The Stat_Type class is an enumerator class. It defines symbolic constants that represent the scope of a statistic being registered.

Table 26-9 Stat_Type Constants

Variable	Type	Value
Local	int	0
Global	int	1

Modeling Web Services Dynamically

This section describes how to model web services using Transaction Whiteboard model files and special “discovery nodes” (UDDI nodes, in web-services terminology). Using this workflow, you can create scenarios in which the requesting node determines the end node dynamically using a discovery node during a simulation, rather than having the requesting node determine the end node statically (that is, using destination preferences).

Workflow Description

To create a dynamic web-services scenario, you must do the following:

1) Create a new Transaction Whiteboard model file—or use an existing file—that defines the discovery behavior (we will refer to this as the *discovery file*). This client models the communication patterns between the node that initiates the discovery and the discovery node itself. This file also models the logic used to determine the end node.

2) Create a new Transaction Whiteboard model file—or use an existing file—that defines the application behavior (we will refer to this as the *application file*). This file models the application behavior between the client and the end node; this end node will be “discovered” during the simulation.

3) Create or modify a network scenario so that it contains at least one of each of the following: a client node, an end node, and a discovery node (UDDI server).

4) In the Deploy Applications dialog box, configure the destination preferences (to map the client’s destination to the discovery node) and the service registration (to register the end node with the discovery node).

Step 1: Define the Discovery File

To simulate a discovery scenario, you need a Transaction Whiteboard model that contains a Client and a Discovery Node tier. The file should specify the messages exchanged between the tiers and the logic by which the Discovery Node tier determines the end node.

Note—If you create your own discovery file, the initiating tier in the Transaction Whiteboard model must be called “Client.” During a discrete event simulation, the simulation kernel maps this tier to the node that initiates the conversation.

The model library includes a Transaction Whiteboard model file (Simple_Discovery.aed.m) that models a simple type of discovery logic, in which the Discovery Node selects an end node based on the relative weights of the candidate nodes. The transaction in this file is executed as a child task of the main application, and consists of the following events:

1) Client (initiating) tier sends a 1000-byte request to the Discovery Node tier.

2) Discovery Node tier looks at the "Service Registration" attribute on the node that is acting as the discovery node, and uses this information to map a network node to the unknown tier.

   The discovery logic is specified in the functions `resolve_tier()` and `get_actual_name_for_tier()`; to see these functions, choose Scripting > Function Block.

3) Discovery Node tier returns a 1000-byte response to the Client tier.

Step 2: Enable Discovery in the Application File

Before configuring the discovery node in the network scenario (as described in the following steps), you must enable discovery in the Transaction Whiteboard model file that models the transaction (such as an FTP download) between the initiating node and the end node. To do this, perform the following steps:

1) Open the Transaction Whiteboard model file that models the application transaction.

2) Choose Scripting > Deployment Options…

3) Make sure that the "Allow discovery using a 'Discovery Node'" option is selected.

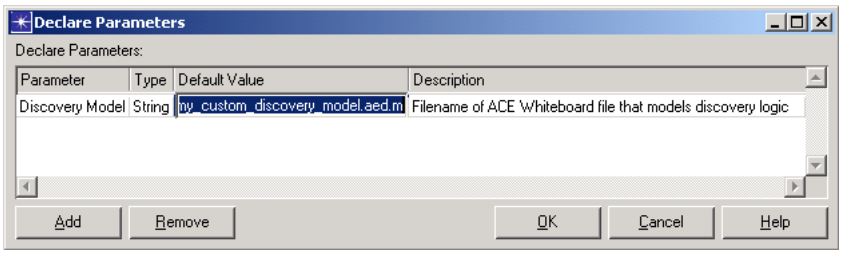
Note—When discovery is enabled in a Transaction Whiteboard application file, it uses the Transaction Whiteboard discovery file Simple_Discovery.aed.m by default. If you want to use a different discovery file, the Transaction Whiteboard application file must include a "Discovery Model" parameter that specifies the name of the discovery file.

To create this parameter in the application file, do the following:

1) Choose Scripting > Declare Parameters.

2) Create a parameter called "Discovery Model" and specify the file name of the Transaction Whiteboard discovery file, as shown in the following figure.

Figure 26-11 Specifying a "Discovery Model" Parameter



Step 3: Create the Network Scenario

The discovery scenario should contain the following:

- A client node that initiates the transaction
- A discovery node. If discovery is enabled in the Transaction Whiteboard model file (as described in the previous step), you can use the eth4_slip4_discovery_node model found in the AppTransaction Xpert object palette.
- One or more end nodes that are possible candidates for the discovery node

Step 4: Deploy the Discovery Application

After you specify the network topology, you can deploy the discovery application. In the Project Editor, choose Protocols > Applications > Deploy AppTransaction Xpert Application on Existing Network > as Discrete Traffic.

Step 5: Configure the Discovery Node

The easiest way to configure the discovery node is by opening the Deploy Applications dialog box (Protocols > Applications > Deploy Defined Applications). There are three sets of configurations you need to verify: application deployment, destination preferences, and service registration. You can change between the different configuration sets using the pull-down menu in the top-right corner of the dialog box.

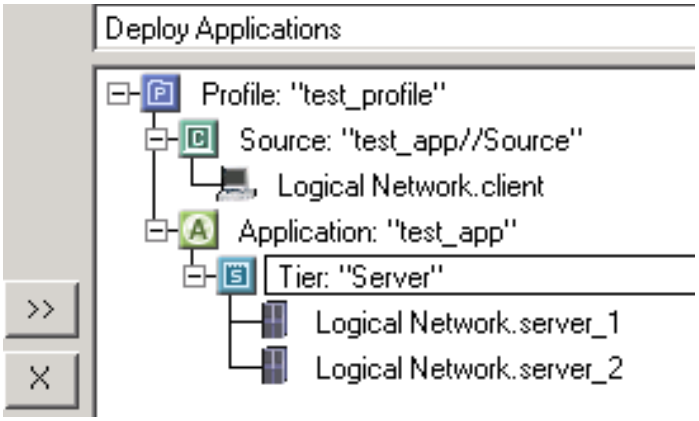
The following examples show the settings for a network that contains a client, a discovery node, and two destination nodes. The discovery node uses the Simple_Discovery model to choose between the two destinations based on their relative weights.

Application Deployment Settings

In the Deploy Applications view, make sure that:

- The initiating node—that is, the node that initiates the transaction—is assigned to the initiating tier of the application
- The destination nodes are assigned to the tier that acts as the destination tier

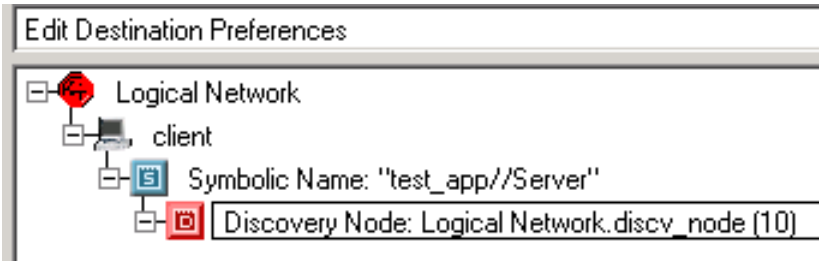
Figure 26-12 Application Deployment Settings: Example



Destination Preference Settings

In the Edit Destination Preferences view, make sure that the discovery node is assigned to the symbolic name of the destination tier.

Figure 26-13 Destination Preference Settings: Example



Service Registration Settings

Use the Edit Service Registration view to configure the discovery node. In this view, make sure that:

- The destination tier of the application is registered with the discovery node for that application
- The candidate end nodes—the nodes that act as possible candidates for the discovery node—are assigned to the destination tier

Figure 26-14 Service Registration Settings: Example