Transaction Whiteboard: Modeling an Application with Logic Scripts (Advanced)

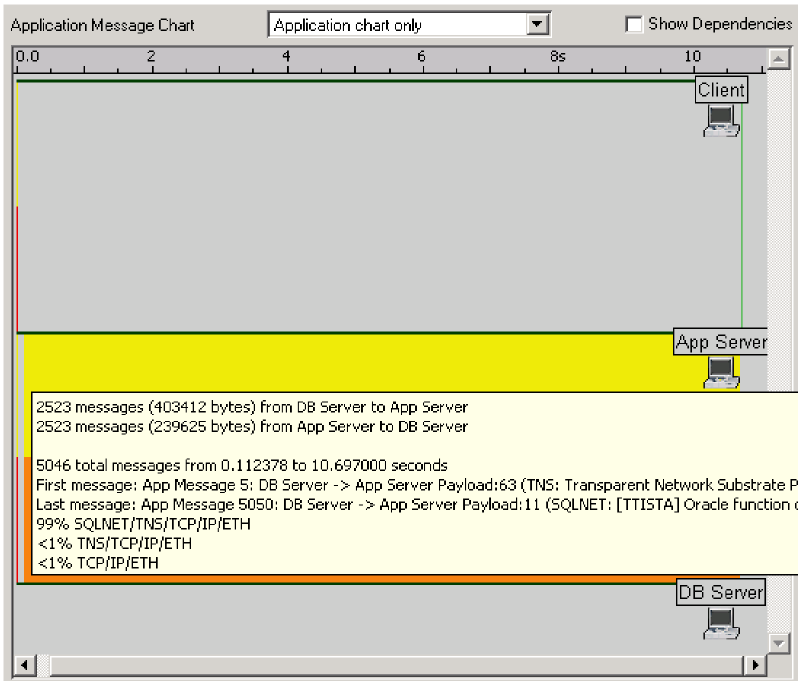
When database applications execute a query, each row that is fetched costs a round trip to the database. Therefore, many database applications employ a *prefetch* value to reduce round-trip times to the server; the prefetch value specifies the number of rows to be fetched from the result set in each trip to the database. The actual prefetch count varies and is configurable. For example, some Oracle interfaces default to 1, while Oracle's JDBC defaults to 10.

In this tutorial, you will use the Transaction Whiteboard Scripting API to model a chatty database application and analyze how changes to the prefetch value impact the expected response time. When deployed, the application server and database will be separated by a MAN (Metro Area Network) with 10 Mbps bandwidth and 1 ms of latency; the client will be accessing the application server via a T1 link.

The objective of this tutorial is to show how to use the Transaction Whiteboard Scripting API to model application performance using different prefetch values.

Examine the Request/Response Pattern when the Prefetch Value is 2

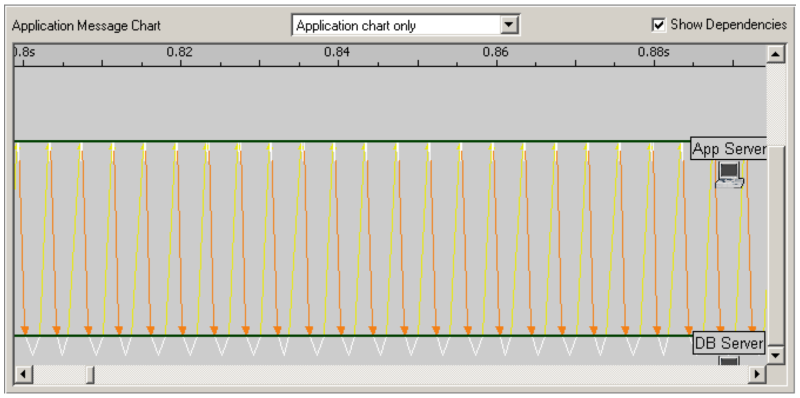
In the figure, notice that more than 5000 application messages are sent between App Server and DB Server.



```
2523 messages (403412 bytes) from DB Server to App Server
2523 messages (239625 bytes) from App Server to DB Server

5046 total messages from 0.112378 to 10.697000 seconds
First message: App Message 5: DB Server -> App Server Payload:63 (TNS: Transparent Network Substrate P
Last message: App Message 5050: DB Server -> App Server Payload:11 (SQLNET: [TTISTA] Oracle function c
99% SQLNET/TNS/TCP/IP/ETH
<1% TNS/TCP/IP/ETH
<1% TCP/IP/ETH
```

In this zoomed-in view, you can see that the traffic between App Server and DB Server consists of a regular request/response pattern.



Model the SQL Logic in Transaction Whiteboard

You will now use Transaction Whiteboard to model the series of request and response messages sent by this chatty database application. For this tutorial, assume that the application exhibits the following behavior:

- Each request to the database is a fixed size (95 bytes).
- Each response from the database is 53 bytes of application overhead, plus 55 bytes per row.
- The database takes a fixed amount of time (0.0018 seconds) to process each request and send the response.
- The application server takes a fixed amount of time (0.00015 seconds) to generate each request.

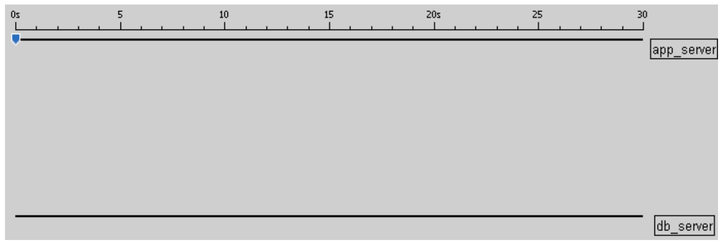
The following procedure describes how to model the request and response messages.

Procedure 13-1   Modeling the SQL Logic

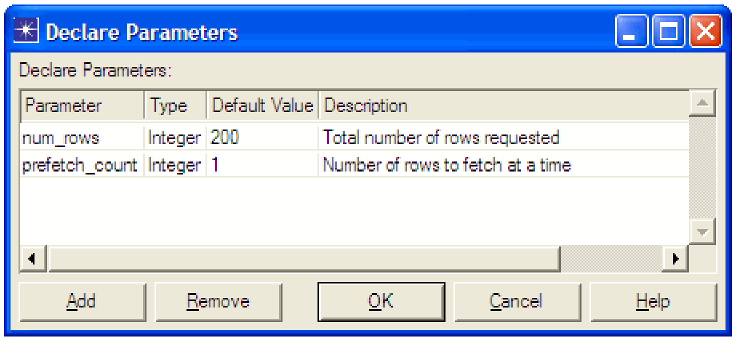
1. Choose **File > Open Model > Transaction Whiteboard…**
2. In the **Transaction Whiteboard Startup Wizard**, select **Open Existing Transaction Whiteboard file** and click **Next >**.
3. Select the file **tutorial_sql_backend** in <*reldir*>\sys\examples\AppTransaction Xpert\examples and click **Open**; the file opens in Transaction Whiteboard.

   Note—*<reldir>* is the release directory where AppTransaction Xpert is installed. (In the Windows environment this is typically C:\Program Files\OPNET\<*release number*>.)

   In the **Data Exchange Chart**, notice that this file contains two tiers and no messages, as shown in the following figure.



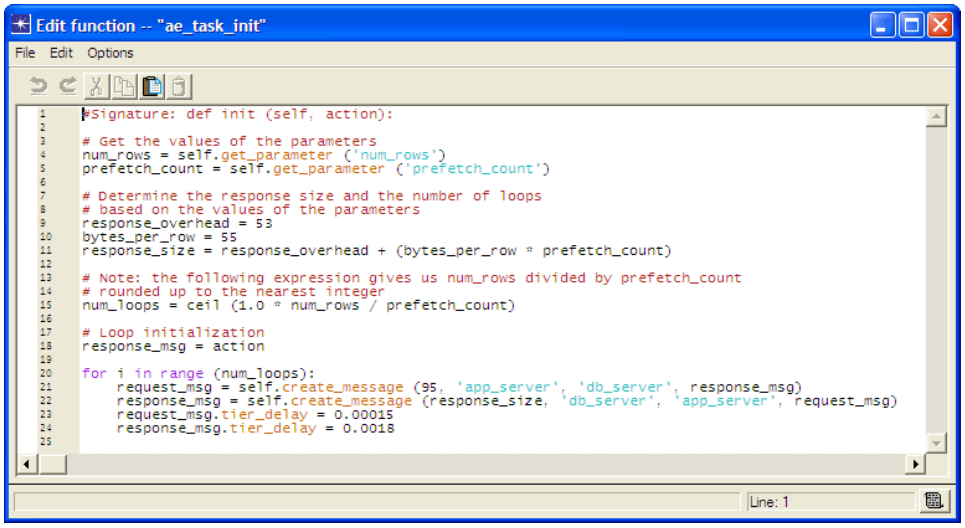
4 Choose **Scripting > Declare Parameters…** Notice that there are two declared parameters. The first parameter, num_rows, specifies the total number of rows requested. The second parameter, prefetch_count, determines how many rows the database sends back for each request. Click **Cancel** to close the Declare Parameters dialog box.



Parameter	Type	Default Value	Description
num_rows	Integer	200	Total number of rows requested
prefetch_count	Integer	1	Number of rows to fetch at a time

5 Choose **Scripting > Initialization Block**.

➥ The following Python code appears in an edit pad.



```
#Signature: def init (self, action):

# Get the values of the parameters
num_rows = self.get_parameter ('num_rows')
prefetch_count = self.get_parameter ('prefetch_count')

# Determine the response size and the number of loops
# based on the values of the parameters
response_overhead = 53
bytes_per_row = 55
response_size = response_overhead + (bytes_per_row * prefetch_count)

# Note: the following expression gives us num_rows divided by prefetch_count
# rounded up to the nearest integer
num_loops = ceil (1.0 * num_rows / prefetch_count)

# Loop initialization
response_msg = action

for i in range (num_loops):
    request_msg = self.create_message (95, 'app_server', 'db_server', response_msg)
    response_msg = self.create_message (response_size, 'db_server', 'app_server', request_msg)
    request_msg.tier_delay = 0.00015
    response_msg.tier_delay = 0.0018

```

Notice that this code:
(a) accesses the values of two declared parameters,
(b) uses the values to determine the response size and the number of loops, and
(c) sends a request and a response message from within a “for” loop.

Also notice that when num_rows is set to 200 and prefetch_count is set to 1, the loop will be executed 200 times, creating a total of 400 messages.

Close the window using the “X” button in the upper-right corner.

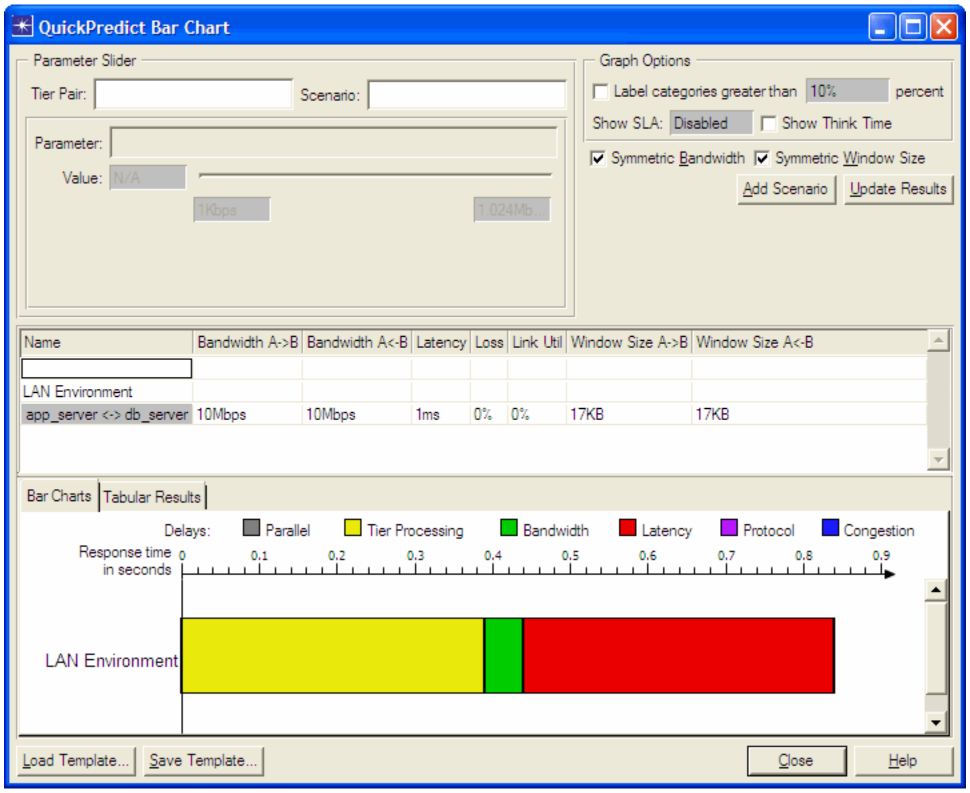
6 Click the **QuickPredict** tool button to open QuickPredict.

7 Click the **Bandwidth** field and change the value to **10 Mbps**.

8 Click the **Latency** field and change the value to **1 ms**.

9 Click **Update Results**.

➥ The QuickPredict window should look like the following figure.



Name	Bandwidth A->B	Bandwidth A<-B	Latency	Loss	Link Util	Window Size A->B	Window Size A<-B
LAN Environment							
app_server <-> db_server	10Mbps	10Mbps	1ms	0%	0%	17KB	17KB

Notice that the latency delay reported by QuickPredict was 400 ms. This is due to the fact that 400 messages are being sent, each of which incurs 1 ms of latency.

10 Click **Close** to close QuickPredict.

11 Choose **File** > **Close** to close the model.

End of Procedure 13-1

In this section, you examined a Transaction Whiteboard model that modeled the request/response pattern of a database application using messages dynamically created in the Initialization Block. For this chatty database application, you saw that the number of rows requested divided by the prefetch count equaled the number of request/response patterns, where each request/response pattern consists of two messages. Since each subsequent message was in the opposite direction, you can tell that latency delay could potentially be a performance bottleneck. Now, you will include the chatty database behavior in a Transaction Whiteboard model of the entire three-tier database application.

Model the Application in Transaction Whiteboard

Procedure 13-2   Modeling the Application

1 Choose **File > Open Model > Transaction Whiteboard…**

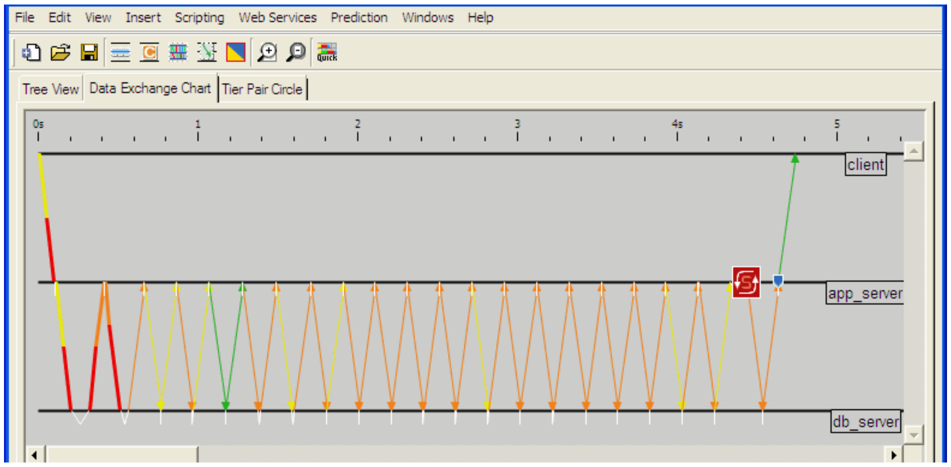
2 In the **Transaction Whiteboard Startup Wizard**, select **Open existing Transaction Whiteboard file** and click **Next >**.

3 Select the file **tutorial_sql_prefetch** in **<*reldir*>\sys\examples\AppTransaction Xpert\examples**, then click **Open**.

➥ The file opens in Transaction Whiteboard.

4 Choose **File > Save As…**, then save the file as **<*initials*>_tutorial_sql_prefetch**.

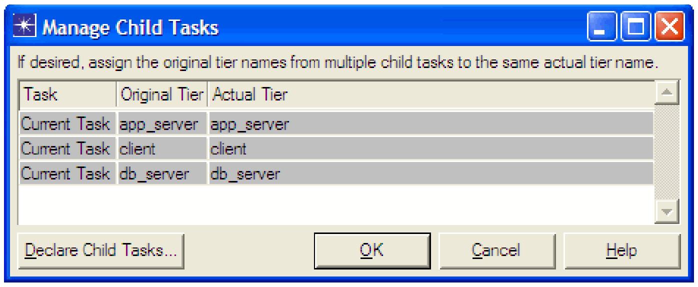
Notice that this three-tier Transaction Whiteboard model contains a logic script; you are going to use that logic script to invoke or *chain in* the database request file that you just examined. In other words, the application behavior in the file that we just examined will now be dynamically added to this file.



To invoke a Transaction Whiteboard file from a logic script, you must declare the child task.

5 Choose **Scripting > Manage Child Tasks**.

➥ The Manage Child Tasks dialog box appears.



If desired, assign the original tier names from multiple child tasks to the same actual tier name.

Task	Original Tier	Actual Tier
Current Task	app_server	app_server
Current Task	client	client
Current Task	db_server	db_server

6 In the Manage Child Tasks dialog box, click **Declare Child Tasks…**

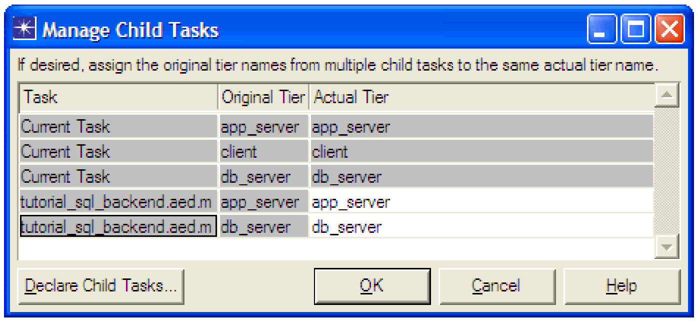
➥ The **Declare Child** AppTransaction Xpert **Tasks** dialog box appears.

7 Since you are declaring a Transaction Whiteboard file, click **Continue…**

➥ The **Declare Child Transaction Whiteboard Tasks** dialog box appears.

8 Select the file **tutorial_sql_backend** under <*reldir*>\sys\examples\AppTransaction Xpert\examples, and then click **OK**.

Notice that the file **tutorial_sql_backend.aed.m** is listed in the **Manage Child Tasks** dialog box.



If desired, assign the original tier names from multiple child tasks to the same actual tier name.

Task	Original Tier	Actual Tier
Current Task	app_server	app_server
Current Task	client	client
Current Task	db_server	db_server
tutorial_sql_backend.aed.m	app_server	app_server
tutorial_sql_backend.aed.m	db_server	db_server

9 In addition to declaring the child task, you can use this dialog box to specify how each tier name in the child task relates to the corresponding tier in the current task. In this example, the tier names in the child task already correspond directly to tier names in the current task.

10 Click **OK** to close the dialog box.

You will now chain in the child task in the logic script.

11 In the **Data Exchange Chart**, double-click on the **Logic Script** icon.

12 Add the following code to the logic script. The first line creates a Python dictionary that contains two entries. The second line invokes the task tutorial_sq1_backend.aed.m. The script invokes this task with num_rows set to 2000 and prefetch_count set to 2.

```
#Signature: def <funcname> (self, action):

child_parameters = {'num_rows':2000, 'prefetch_count':2}
self.invoke_child_task (action,
    'tutorial_sql_backend.aed.m', True, child_parameters)
```

13 Choose **File > Commit** to save the logic script.

14 From the Transaction Whiteboard window, choose **File > Save** to save the file.

End of Procedure 13-2

In this Transaction Whiteboard file, you declared a child task, and then chained in the child task as part of a logic script. Notice that you were able to pass values for each of the declared parameters to the child task.

Use QuickPredict to Determine the Expected Response Time

Procedure 13-3 Determining the Expected Response Time (with QuickPredict)

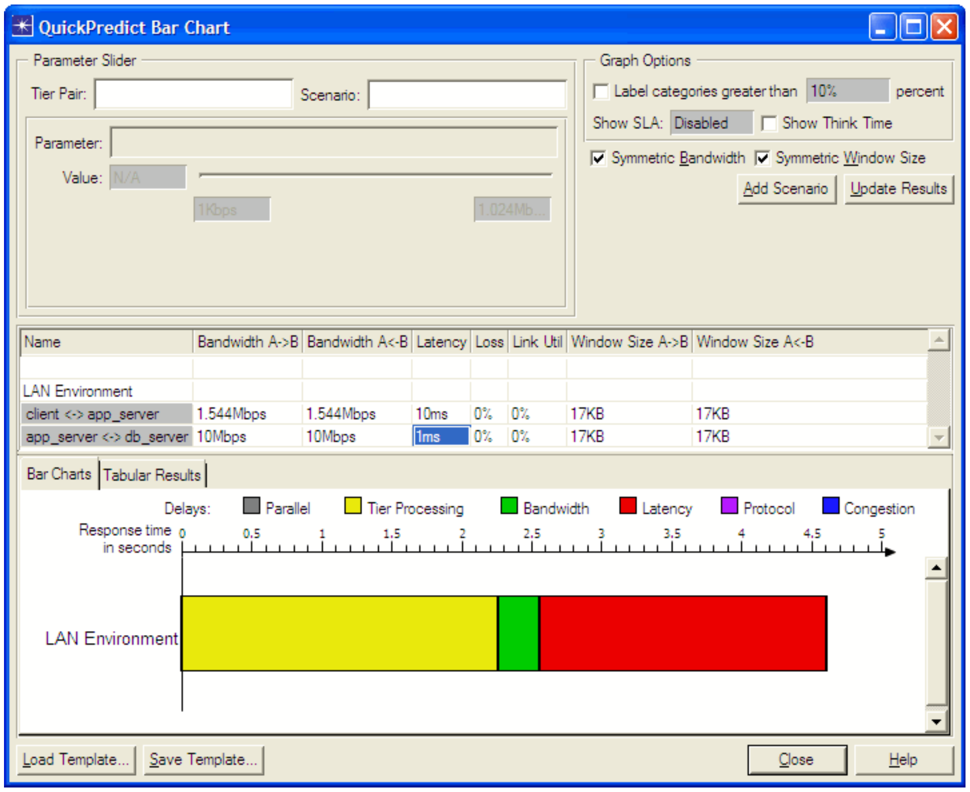
1 Click the **QuickPredict** tool button to open QuickPredict.

2 Set the **Bandwidth** for the first tier pair, client<–>app_server, is **1.544 Mbps**. Set the **Latency** for the first tier pair to **10 ms**.

3 Change the **Bandwidth** for the second tier pair, app_server<–>db_server, to **10 Mbps** and the **Latency** to **1 ms**.

4 Click **Update Results**.

➥ The QuickPredict window should look like the following figure.



Value: N/A
1Kbps
1.024Mb...

[ ] Label categories greater than 10% percent
Show SLA: Disabled [ ] Show Think Time
[✓] Symmetric Bandwidth [✓] Symmetric Window Size

Name	Bandwidth A->B	Bandwidth A<-B	Latency	Loss	Link Util	Window Size A->B	Window Size A<-B
LAN Environment							
client <-> app_server	1.544Mbps	1.544Mbps	10ms	0%	0%	17KB	17KB
app_server <-> db_server	10Mbps	10Mbps	1ms	0%	0%	17KB	17KB

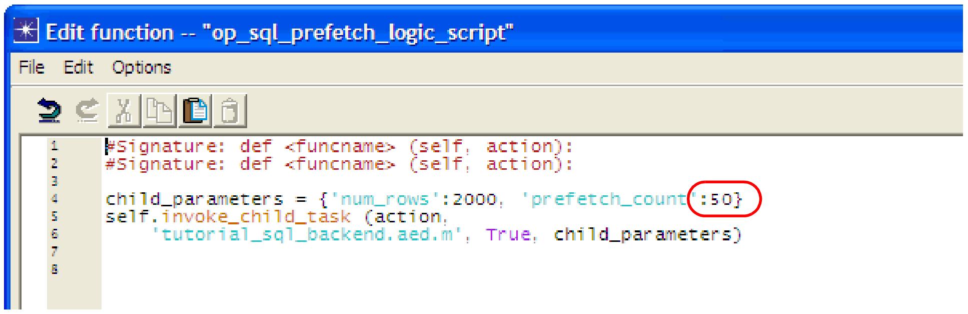
By examining the QuickPredict results, you can see that this application takes over 4 seconds to retrieve 2000 rows from the database.

5 Leave the QuickPredict window open.

You will now increase the prefetch count from 2 to 50.

6 In the **Data Exchange Chart**, double-click on the **Logic Script** icon.

7 Change the **prefetch_count** from 2 to **50**.



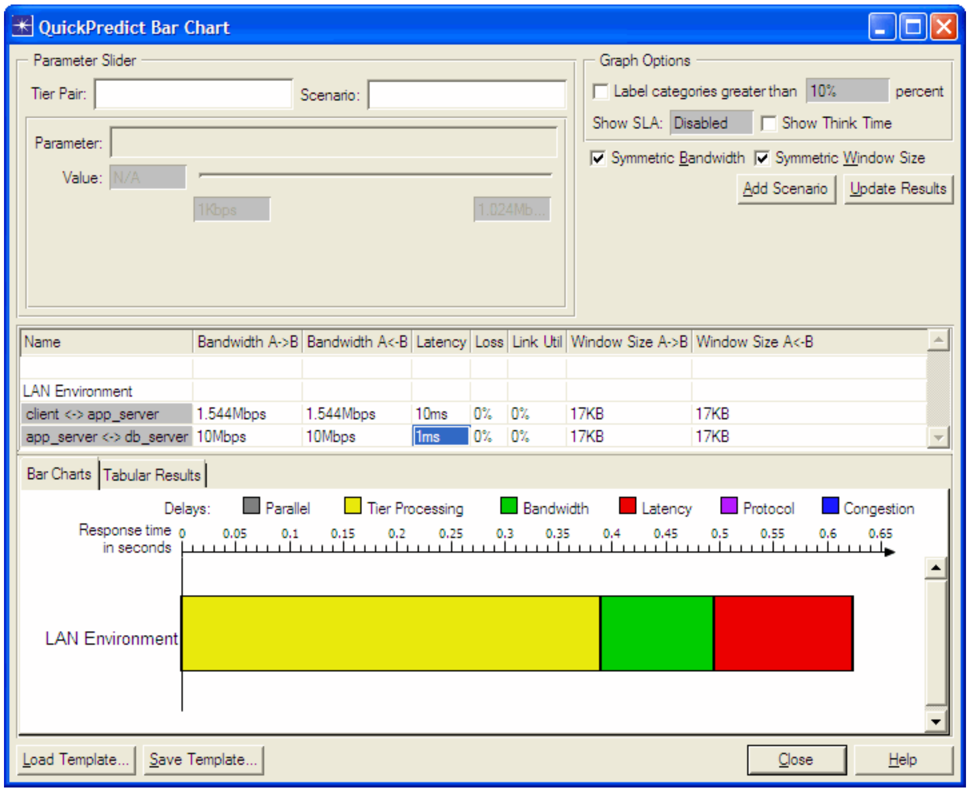
```
#Signature: def <funcname> (self, action):
#Signature: def <funcname> (self, action):

child_parameters = {'num_rows':2000, 'prefetch_count':50}
self.invoke_child_task (action,
    'tutorial_sql_backend.aed.m', True, child_parameters)
```

8 Choose **File > Commit** to save the logic script.

9 In the QuickPredict dialog box, click **Update Results**.

➥ The QuickPredict window should look like the following figure.



Name	Bandwidth A->B	Bandwidth A<-B	Latency	Loss	Link Util	Window Size A->B	Window Size A<-B
LAN Environment							
client <-> app_server	1.544Mbps	1.544Mbps	10ms	0%	0%	17KB	17KB
app_server <-> db_server	10Mbps	10Mbps	1ms	0%	0%	17KB	17KB

End of Procedure 13-3

By increasing the prefetch count from 2 to 50, there is a dramatic decrease in the response time of the application when requesting 2000 rows. Instead of making 1000 requests of 2 rows each, increasing the prefetch count to 50 resulted in only making 40 requests of 50 rows each.

Conclusion

In this tutorial, you used Transaction Whiteboard to model the effects of changing the prefetch count for a chatty database application. First, you created a Transaction Whiteboard model of the chatty database request/response pattern. Then, you invoked this task from within a second Transaction Whiteboard model. By changing the values of the parameters passed to the child task, you saw that increasing the prefetch value resulted in fewer request/response patterns, which reduced both the tier processing delay and the latency delay.

Additional Details

The back-end Transaction Whiteboard file used in this tutorial contained two tiers and no drawn messages. The application messages were created dynamically in the Initialization Block. Alternatively, you could have drawn the request/response pattern and used the `goto_action` method from the Transaction Whiteboard Scripting APIs to create the same behavior.