HTTP transport nodes in WebSphere Message Broker V6

Kevin Braithwaite
Rich Bicheno

August 09, 2006

This article shows you how to use the new HTTP transport nodes in WebSphere Message Broker V6 to add a facade to an existing application, enrich an existing HTTP service, or request an HTTP service from within an existing messaging backbone.

Introduction

Hypertext Transfer Protocol (HTTP) is a widely used standard protocol built on top of the TCP/IP stack for request/reply-based communications. The protocol is of course most commonly used by Web browsers to request pages from a server. With the development of the Enterprise Service Bus (ESB) technology and Service-Oriented Architecture (SOA), HTTP is becoming more widely used as the transport mechanism for invoking services, including Web services. HTTP is simply a protocol for communication between an HTTP client and an HTTP server. It does not let you interact with non-HTTP-enabled services in your enterprise.

The HTTP nodes in IBM® WebSphere® Message Broker V6.0.0.1 (hereafter called Message Broker) help you solve this problem. This article describes the HTTP nodes, shows you how to configure and use them, and discusses performance and scalability issues to consider when implementing message flows that use the HTTP nodes, including performance characteristics of several usage scenarios. For information on implementing Web services using WSDL, see Web services support in WebSphere Message Broker V6.

More about HTTP

The HTTP protocol was designed to be simple, fast, extensible, and compatible with previous versions. Although data retrieval using HTTP is efficient, the way connections are handled between client and server is not efficient where there are multiple requests for data. A new connection is required for each request/response pair, imposing a significant overhead on communications. To overcome this problem, HTPP 1.0 or later supports persistent connections between client and server, with multiple request/response pairs sent via the same connection. This function is available via the connection: keep-alive header and became the default in HTTP 1.1, with the client and server required to explicitly close the connection using the connection: close

header. Persistent connection is very useful in some situations, such as when trying to repeatedly invoke a service, as a new socket no longer needs to be created for each request, dramatically improving performance.

HTTP is becoming a popular method for invoking services over TCP/IP, using standards such as Simple Object Access Protocol (SOAP) for the content of the request. Since HTTP and SOAP are widely used standards, this method is a good way to invoke a Web service, though you can also invoke services with other technologies and standards such as messaging (using WebSphere MQ for example), or SOAP over JMS. A key advantage of HTTP is that it is often not blocked by firewalls, enabling applications that use HTTP to be more available.

Integrating HTTP within the enterprise

A major advantage of HTTP is that it is a well-defined and vendor-neutral standard. HTTP by itself, however, is not suitable for all enterprise Web service invocations, because most large enterprises have many applications and services that are not HTTP enabled, and data from these applications often needs to be transformed before it can be understood by other enterprise applications. Yet many of these existing applications are well-defined, reliable, and high-performing, and therefore they often need to be integrated with the rest of the enterprise, instead of being re-architected or rebuilt.

HTTP transport nodes

Message Broker supports many different protocols, including:

- HTTP
- WebSphereMQ
- WebSphereMQ Telemetry Transport
- Multicast
- WebSphereMQ Real-time
- JMS Transport

HTTP transport nodes enable Message Broker to receive data from any HTTP client and send data back to that client. In addition, the nodes provide the ability to issue an HTTP request as an HTTP client.

The HTTP nodes act simply as a way of getting data into Message Broker. Once the data has arrived, all of the functionality normally associated with Message Broker is available. For example:

Data computations and transformations using:

- ESQL
- Java Computer nodes
- Enrichment of data from external sources
- Routing

There are three HTTP transport nodes in Message Broker:

HTTPInput node

Like other input nodes such as MQInput, the HTTPInput node obtains data from the client using HTTP as the communication protocol as opposed to WebSphere MQ or JMS. HTTPInput node supports several message domain formats for the incoming data:

- MRM
- XML
- XMLNS
- XMLNSC
- JMS
- JMSStream
- IDOC
- MIME
- BLOB

Once the data is received, HTTPInput node parses the data into the internal Message Broker message tree, enabling all normal Message Broker functionality to be used in the remainder of the message flow. An HTTPInput node must be in the same flow as an HTTPReply node, or it must pass the message to another flow containing an HTTPReply node, such as via an MQOutput node, because HTTP is a request/reply protocol and therefore a connecting client always expects a reply of some kind. HTTPInput node supports both HTTP and HTTP over SSL (HTTPS).

HTTPReply node

HTTPReply node sends a response back from Message Broker to the HTTP client that originally invoked the flow. The response is matched to the original request. The HTTPReply node must be in a flow that contains an HTTPInput node, or the original message must have been received from a flow that contained an HTTPInput node. The entire message tree is used as the body of the response.

HTTPRequest node

HTTPRequest node can be used within a message flow to enable Message Broker to invoke an existing HTTP-based service. The request can be made up of the entire message flow message tree or specified parts of it. You can augment the contents of the original input message (the request sent to the HTTP service) with the response from the HTTP service to create a new message before propagating it to subsequent nodes in the flow. HTTPRequest supports several different message domain formats for the incoming data:

- MRM
- XML
- XMLNS
- XMLNSC
- JMS
- JMSStream
- IDOC
- MIME

- BLOB

HTTPRequest supports both HTTP and HTTP over SSL (HTTPS).

Node configuration

More information on configuring HTTP transport nodes.

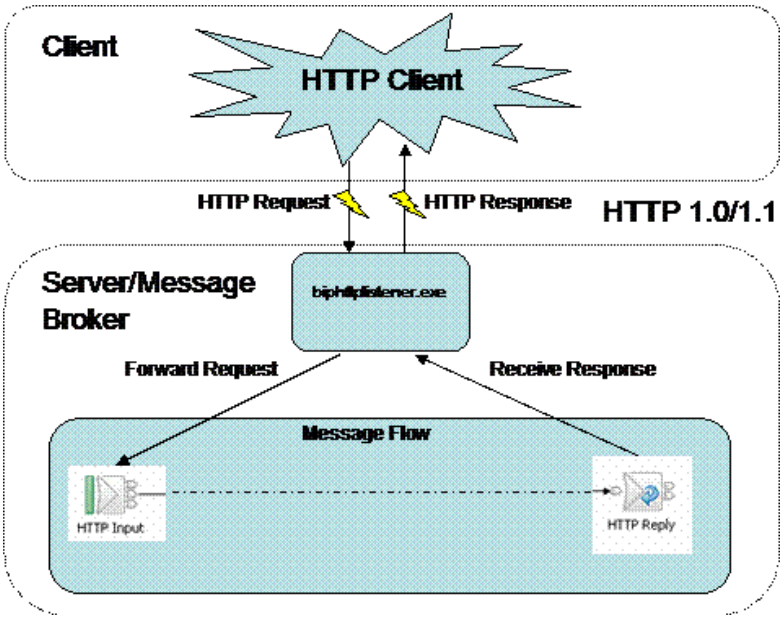
HTTP connectivity

The previous section described the HTTP nodes. This section explains how HTTP communication is handled between a Message Broker message flow and a client that may invoke the message flow, or an HTTP-based service that can be invoked from within a message flow.

Broker as server

The client must be capable of using the HTTP protocol. Receipt of HTTP requests in Message Broker is handled through a listener process called `biphttplistener`. Figure 1 below shows how this interaction takes place. The HTTP request issued by the client is received by the Message Broker listener process. The message is then passed to the HTTPInput node and processing within the message commences. Upon completion of the message flow, the HTTPReply node issues a response, which is sent to the HTTP client through the Message Broker HTTP listener process.

Figure 1. HTTP request and response



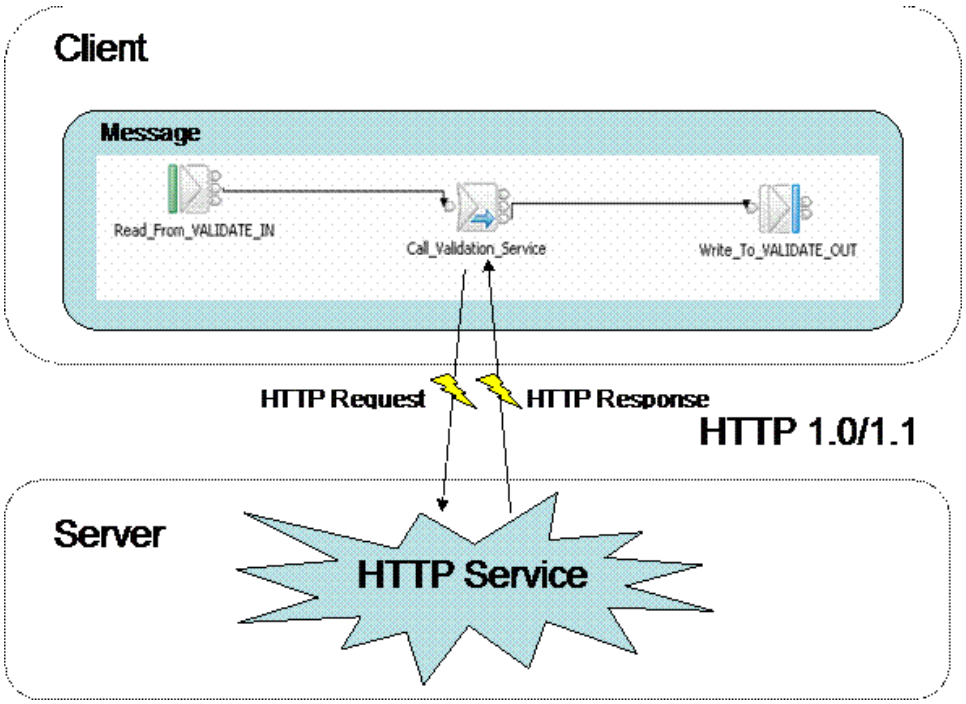
There is a single multithreaded (configurable) `biphttplistener` process for each Message Broker instance. The `biphttplistener` process is resilient -- if it fails, it automatically restarts. A single listener serves all flows containing an HTTPInput node deployed to a single broker. You can, however, filter the incoming HTTP requests to different flows based on the incoming URL by setting properties on the HTTPInput node. You can use the entire URL or a pattern. For example, if

the URL that you want is `http://<hostname>[:<port>]/[<path>]`, then specify either `/<path>` or `/<path fragment>/*`, where `*` is a wild card.

Broker as client

When the interaction is with an HTTP-based service, the processing sequence is different and involves a different processing node. Figure 2 shows how a message flow invokes an HTTP-based service. In this case, the message flow is acting as a client using the HTTPRequest node:

Figure 2. HTTP-based service



Technology benefits and business value

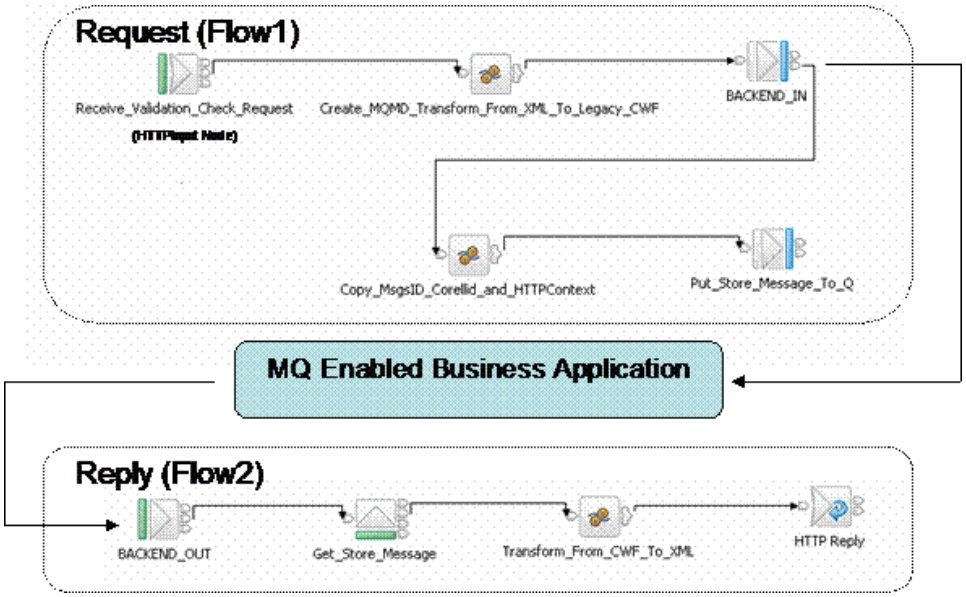
So far we have focused on what the HTTP Transport nodes are and how you can configure and use them within a message flow. This section uses some common usage scenarios to show how you can use the nodes to add value to your existing enterprise messaging infrastructure. Scenarios featured in this section will also be the scenarios featured within the section Performance characteristics and scalability.

Facade an existing application

When implementing an SOA, many of the services that you want make available across your enterprise will probably not be HTTP enabled. A number of applications may perform key business processing and must be used across other areas of the enterprise. By HTTP-enabling such applications or pieces of function using the HTTPInput and HTTPReply nodes, you can make them more easily accessible. This article discusses two types of applications that you can facade using WebSphere Message Broker -- WebSphere MQ enabled and non-WebSphere MQ enabled, such as a DB2 stored procedure or a CORBA application.

For WebSphere MQ enabled applications, a typical scenario involves a request/reply application:

Figure 3. Request/reply application



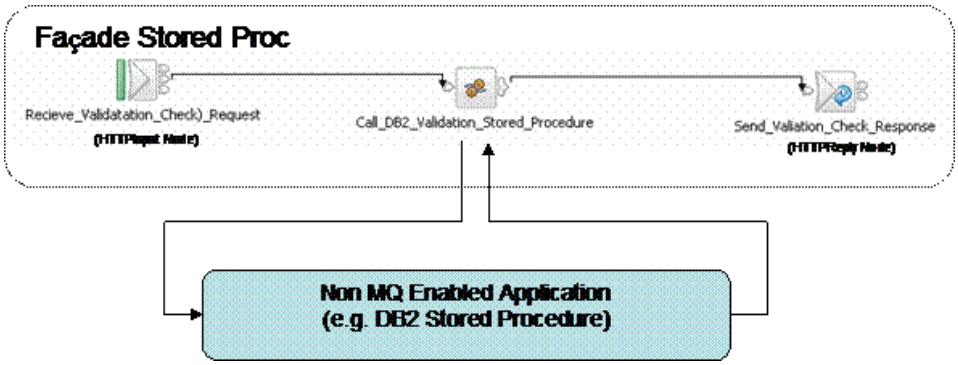
In this scenario, you start with an HTTPInput node to construct `Flow1`, which receives a request from an HTTP Client. This request is for a WebSphere MQ enabled service. The request is transformed into the appropriate format, including WebSphere MQ headers, and placed on the queue that the business application reads from. The request identifier of the incoming message is also stored to enable coordination of replies from the application back to the requesting HTTP Client by `Flow2`.

The business application performs its operation and places the resulting message on its output queue, where it is picked up by the Reply message flow, `Flow2`, through a standard MQInput node. The request identifier is retrieved for this message, the message is transformed into the appropriate format including HTTP headers, and then sent back to initial requesting HTTP Client via the HTTPReply node.

This process is straightforward and the created message flows are simple. Once the HTTP Request has entered Message Broker, all the normal functions are available. As demonstrated, you can do conversion between message formats -- XML to Custom Wire Format -- and thereby communicate with an array of external applications.

Not all applications or pieces of business processes that you want to expose as services will be WebSphere MQ enabled. Examples include a DB2 stored procedures or a CORBA application. For non-WebSphere MQ enabled applications, imagine a "straight-through" scenario:

Figure 4. Non-WebSphere MQ enabled application



In this scenario, the message flow receives a request from an HTTP client and the requested application, a DB2 stored procedure, is called from within a Compute node. Parameters for the application are extracted from the incoming HTTP Request message and converted into the appropriate format. You make a synchronous call to the application and receive a response, which is built into the original HTTP request message and sent back to the HTTP client as an HTTP response. Again, this is a very simple message flow but very powerful.

The HTTP client is only aware of a synchronous call. The contents of the message flow may be either synchronous or asynchronous, which is transparent to the requesting client and thus offers considerable flexibility in the way the service is composed.

In addition to the composition of the message flow, you need to consider the duration of the connection between the HTTP client/server and the message flow. You can configure the HTTP connection as either short-lived or long-lived.

The short-lived, non-persistent connection is best when many (hundreds or thousands) of clients want to make individual requests and then disconnect. You should not maintain large number of connections in this case because of the memory and other resources required.

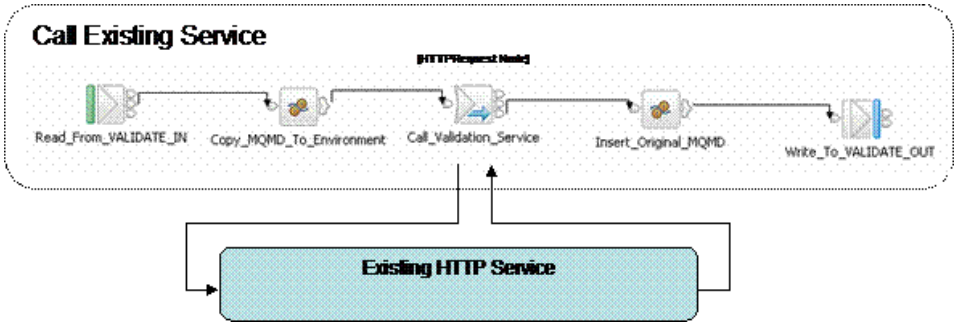
The long-lived, persistent connection is best when a small number of requesting applications need high volume message throughput. Long lived connections avoid the overhead of session creation and deletion. This connection is common when a concentrator application funnels requests to a message flow from a large number of users.

Both short- and long-lived connection types are supported in Message Broker V6.0.0.1. Configuration details are covered below under Performance and tuning.

Issue an HTTP Request from within the messaging backbone

Another common usage scenario is invoking services that are already HTTP enabled. You can do this using the HTTPRequest node:

Figure 5. HTTP-enabled services

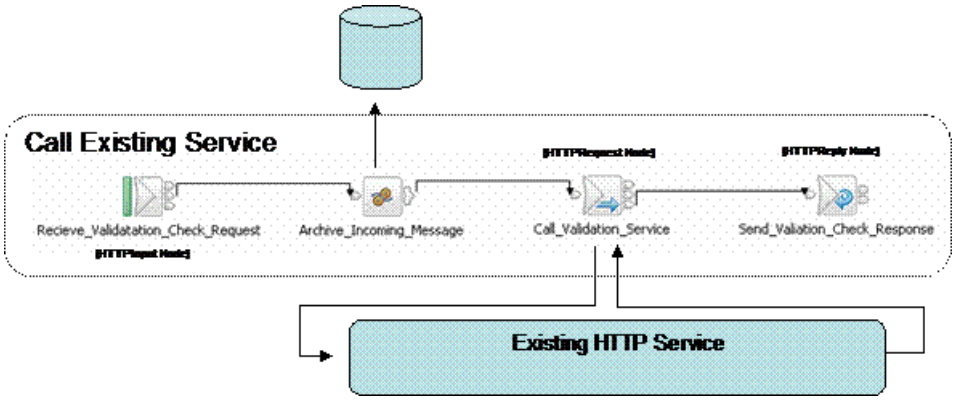


In this scenario, you construct a message flow to read a message from a WebSphere MQ queue via an MQInput node. You store the MQ headers before making a synchronous call to an HTTP service via an HTTPRequest node, using the body of the MQ message as the body of the request (this option is configurable). You receive the response from the HTTP service and use it along with the original stored MQ headers (again, this option is configurable) as the response message written to the WebSphere MQ queue. As above, the message flow is simple to construct and all normal Message Broker functions are available.

Facade an existing HTTP Service

You have seen that you can facade an existing business application as HTTP services, and also call existing HTTP services from within the messaging backbone. An additional scenario is to enrich existing HTTP services, in which you want to invoke an existing HTTP, but also add value to it via Message Broker function. You can do this using the HTTPInput node, HTTPRequest node, and HTTPReply node:

Figure 6. Call existing service



In this scenario, you construct a message flow to receive a request over HTTP from an HTTP client, which is modified to issue a request to the URL of Message Broker instead of the URL of the existing HTTP Service. Archive the HTTP request into a database for audit purposes before forwarding the original request to the existing HTTP service, then forward the response from the HTTP service back to the HTTP client. This scenario demonstrates the auditing of incoming requests, but it does not have to be restricted to such usage. You can use any Message Broker function to enrich an existing HTTP service. This scenario is not included in the Performance

characteristics and scalability section below as it is really a simple extension of the three previous scenarios.

Load balancing and failover

To build a system that can process a high volume of messages and provide high availability, you need to determine the availability level required and whether it is acceptable to have data marooned if a server or message broker fails. Remember, it may not be just the original request that is held up, but also any intermediate processing running on the failed server. The two main availability approaches are load balancing and automatic failover of processing components such as Message Broker.

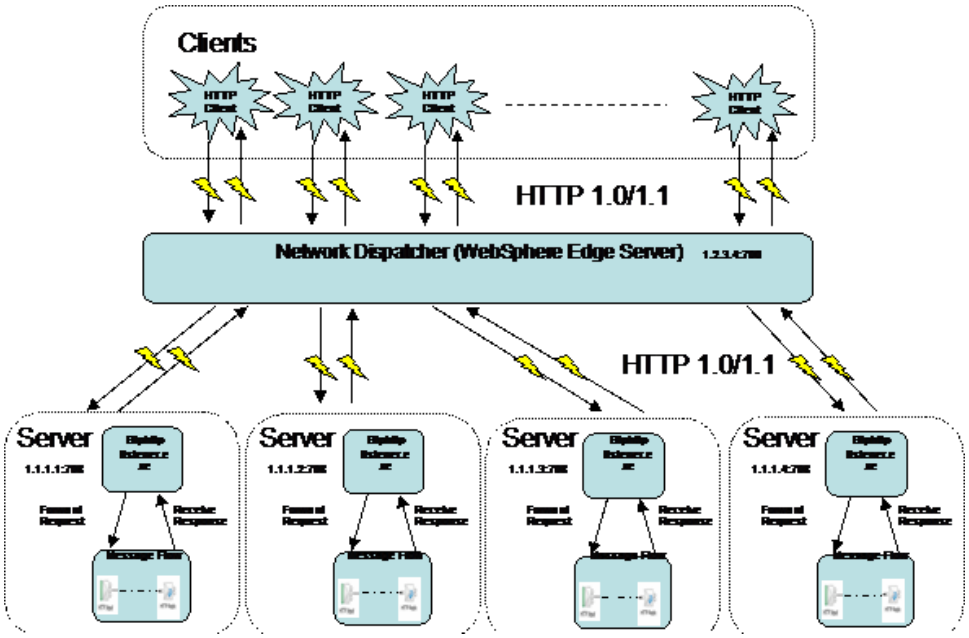
Load balancing is the distribution of incoming work across a pool of available servers or message brokers so that a higher volume of work can be completed. Load balancing can increase system availability by routing requests around failed or poorly performing message brokers, provided that the load balancer monitors the health of each of the servers via health monitoring probes or "heartbeat processing." Here are two approaches for increasing availability:

Load balancing with a pool of servers
: Even if one or more servers fail it will be possible to continue to provide a service using the other servers in the pool. However in the event of a server failing there is the likelihood that some data will become marooned on the failed server .This would remain so until the server was restarted. If the problem were hardware related this could take some time to recover from.

using software such as HACMP that is able to restart a failed server automatically on another machine
: This has the advantage that no data will be marooned. There will be a short delay while the server is restarted one another system, but once it is restarted the original data is still accessible. With WebSphere Message Broker V6 you can use HACMP to restart a failed broker on another system. This approach has been used widely in the past where the incoming work is from WebSphere MQ clients. It works equally as well when the incoming work is from an HTTP client. You should bear in mind though that the incoming request is not recoverable unless it is logged in some way, this is wholly consistent with the nature of the transport protocol.

The remainder of this section focuses on the use of different load balancing solutions and discuss the merits of them. Software to provide high availability, such as HACMP is not discussed any further. Here is a diagram of an environment that can provide load balancing and availability (but no recovery):

Figure 7. Load balancing and availability



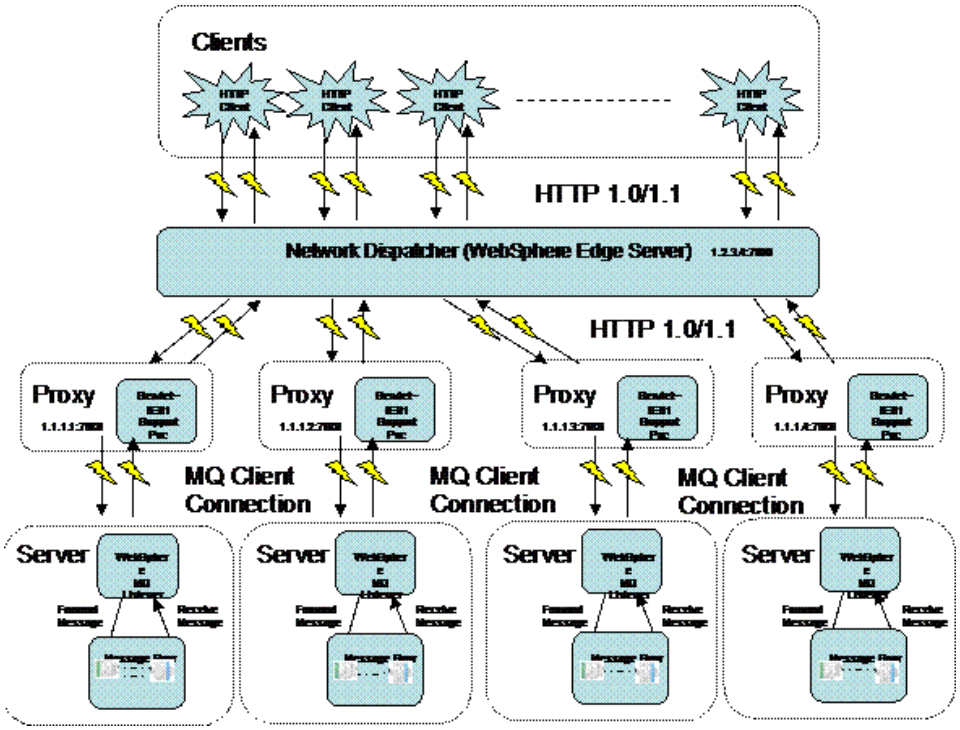
This diagram shows that you can connect multiple HTTP Clients to the same URL and to any number of back-end Message Brokers by using a network dispatcher such as WebSphere Edge Server Load Balancer. The number of server instances is variable, which makes it easy to increase message throughput capacity using horizontal scaling.

Using the network dispatcher lets you distribute workload across all connected brokers. The mechanism for doing this is configurable -- for example by rotating through a check of server response times. Using a network dispatcher also adds failover capabilities by verifying whether the HTTP listener port is functioning on each of the connected servers. It tries to establish a TCP connection from the dispatcher to each of the servers, and if it fails, it marks that server as down stops forwarding requests to that broker's HTTP listener.

With WebSphere Edge Server, more advanced health checks are available using pre-packaged advisors. You can also write custom advisors to implement your own health checks.

An alternative architectural approach is shown in Figure 8:

Figure 8. Moving HTTP connection processing away from broker by using a proxy



This setup is similar to the one in Figure 7, but moves the HTTP connection processing away from the broker by using a proxy. The proxy is a servlet engine running inside an application server such as WebSphere Application Server and acting just like the Message Broker internal process in Figure 1. The servlet engine is available as a Category 3 SupportPac.

As shown in the diagram above, a proxy servlet can service only a single Message Broker. You can, however, have many proxy servlets connecting to a single Message Broker.

The Network Dispatcher is used purely for distributing the incoming HTTP requests. It is not a prerequisite to the proxy servlet. The main reasons for using a proxy servlet architecture is its ability to support a greater number of concurrent connections than you can with the default listener. By using multiple copies of the proxy servlet you can support more concurrent users. For details on setting the number of concurrent connections, see Performance considerations and tuning below. Message Broker can support about 1200 concurrent connections to a broker biphttplistener process. Of course if you need to support fewer than 1200 concurrent sessions, then you do not need to use the Proxy Servlet in this role.

Using transactions with HTTP

HTTP messages are always non-persistent, and have no associated order

HTTP messages are non-transactional. However, if the message flow interacts with a database or another external resource such as a WebSphere MQ queue, these interactions may be performed transactionally if configured to do so. The HTTPInput node provides commit or rollback depending on how the message flow has ended, and how it is configured for error handling (how failure

terminals are connected, for example). If the message flow is rolled back by this node, a fault message is generated and returned to the client. The format of the fault is defined by the Fault Format property.

If an exception occurs downstream in the message flow, and is not caught but is returned to the HTTPInput node, the node constructs an error reply to the client. This error is derived from the exception and the format of the error is defined by the Fault Format property.

Performance considerations and tuning

You can use a standard installation of Message Broker in your enterprise to handle HTTP and issue HTTP requests. In some cases you many need to tune Message Broker to achieve a higher level of throughput. This section discusses the standard configuration and how to modify it.

Persistent connections

Message Broker supports both HTTP 1.0 and 1.1 and can therefore handle persistent connections. When dealing with a small number of requesting applications that need a high volume of message throughput, possibly by using a concentrator application funneling requests to a message flow, you should use a persistent connection to prevent the unnecessary creation and deletion of sockets for each connection. If the HTTP client uses HTTP 1.1 and does not specify the `Connection: close` request header, persistent connections will be used by default. However, there is a configurable property called `maxKeepAliveRequests` that specifies how many requests can be send down a single connection.

The default Message Broker value is 100. Therefore, when an HTTP client connects, it can send 100 requests before Message Broker issues a `Connection: close` and closes the socket. The next request the client sends creates a new socket and again can send up to 100 requests. In many environments this setting may be enough. But when maximum throughput is required, you may need to set this value higher or make it unlimited, in order to achieve the desired throughput.

Here is the command: `mqsichangeproperties MyBroker -b httplistener -o HTTPConnector -n maxKeepAliveRequests -v 0`.

`MyBroker`
: Name of the Message Broker

`-b`
: Component you are changing

`-o`
: Object within the component you wish to change

`-n`
: Name of specific parameter to change

`-v`
: Value

In the above command a value of 0 is specified, which sets the number of requests allowed down a single socket to unlimited. This setting is used for the measurements shown below in the section Performance characteristics and scalability.

As noted in previous sections, you can call HTTP services from Message Broker, which acts like any other HTTP client and can thus create persistent connections. The HTTPRequest node has a configuration option to use keep alive connections, though it is supported only when you select HTTP 1.1. The value of the `maxKeepAliveRequests` parameter on the server that the HTTPRequest node is connecting to determines how frequently a new connection is created.

Concurrent connections

In addition to dealing with a small number of long-lived connections by using persistent connections, you may also have a large number of short-lived connections involving hundreds or thousands of clients that want to make individual requests and then disconnect. The HTTP client can issue an HTTP request with Connection: close specified in its headers, which ensures that a new connection is created for each new request.

You can set the `maxThreads` parameter to determine the maximum number of concurrent connections that Message Broker will accept. But machine resources, such as amount of addressable memory, determine the absolute maximum number of concurrent HTTP client connections that you can make to Message Broker, and these resources may be exhausted before the `maxThreads` limit is reached.

The default value is 250, which means the 251st client that tries to connect will get a Connection Refused response. (Actually, a few connections in excess of the `maxThreads` value may be allowed, because connections can be queued up to the limit set by the `acceptCount` parameter, which is again configurable). In many environments, the default setting may be enough, but in some cases you may need to set this value higher by running this command: `mqsichangeproperties MyBroker -b httplistener -o HTTPConnector -n maxThreads -v 2000`. (For the meanings of the parameters, see above list.)

The above command sets the number of concurrent connections allowed to 2000. This setting is used for the measurements shown below in the section Performance characteristics and scalability.

Increase the heap size of the JVM running within the biphttplistener process to let the JVM allocate more threads for the incoming connections. The default setting for the listener JVM is 192 MB, but you should increase it to 512 MB when dealing with large numbers of concurrent connections. To change the value, modify the Message Broker registry.

Windows®

Under the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\IBM\WebSphereMQIntegrator\2\<BrokerName>\CurrentVersion` create a new String value named `MaxJVMHeapSize`. Set this new String value to the JVM maximum heap size required in bytes. For example: `536870912` for 512 MB.

UNIX®

In the directory `<Broker File Path>/<Broker Name>/CurrentVersion/`, create a new file named `MaxJVMHeapSize`. Set the contents of this file to the JVM maximum heap size required in bytes. For example: `536870912` for 512 MB.

For the changes to take affect, you will to stop and restart Message Broker.

The methods described above increase the specific JVM that the `biphttplistener` process runs within. This JVM is separate from the one for the execution group processes, and therefore cannot be modified using the `mqsichangeproperties` command.

When dealing with large numbers of open sockets (a single socket per concurrent connection) the operating system may limit the number of files that a single process can open at one time. On UNIX, the limit for the number of files that a process can open also applies to sockets, and therefore you need to increase the maximum open file handles setting to reflect the expected number of concurrent connections. To check the current setting, run the command `ulimit -a`. For help in increasing this setting, see your system administrator. Normally you can change it with the command `ulimit -n 4096`.

This setting is only for UNIX. When running with the above parameters, `maxrtheads=2000, JVM=512, ulimit=4096` (where appropriate), 1200 concurrent connections were created using a single Message Broker instance on Solaris, Windows, and AIX, because of the hard limit being reached in WebSphere MQ.

Performance tuning the HTTPRequest Node

By default, the HTTPRequest node uses the platform default for its sockets `tcpnodelay` setting. This setting disables the use of Nagles Algorithm, which is a TCP feature that buffers up small packets into a single large packet for more efficient transmission. This setting can briefly delay the transmission of smaller packets, especially on some platforms. For example, on AIX, it has resulted in poor response time and throughput when sending small messages using the HTTPRequest node. To disable Nagles Algorithm, use this setting:

```
mqsichangeproperties <BrokerName> -e <ExecutionGroupName> -o
  ComIbmSocketConnectionManager -n tcpNoDelay -v true
```

It is much more efficient from a performance perspective to use persistent HTTP connections. To ensure that the HTTPRequest node uses persistent connections, check the box on the node labeled "Enable HTTP/1.1 keep-alive" under "HTTP Settings" on the node properties. The remote end to which you are connecting must also be configured to accept persistent HTTP connections, and this configuration depends on the Web service provider/HTTP Server you are using. By default, the socket used in the request node is recycled every 90 requests. To set the node to use the same socket for an unlimited number of requests, use the command:

```
mqsichangeproperties <BrokerName> -e <ExecutionGroupName> -o
  ComIbmSocketConnectionManager -n  maxKeepAliveRequests -v 0
```

However, even with this setting, the socket will be closed if it is idle for more than four seconds. After you run either of the commands above, you must stop and restart the broker. To verify

whether persistent connections are being used, use the command `netstat -a` on the broker machine to check the connections between it and the remote server. The number of sockets in use should be equal to the number of flow instances -- in other words, one connection per thread.

Performance characteristics and scalability

This section contains throughput results when running a number of test cases. The tests include those discussed in this paper plus a number of high-volume simple test cases. To reduce the number of tests, only the 1K message size was scaled across execution groups.

Windows

Figure 9. Throughput results on Windows

Façade Back End		Scaling for 1K Message Only							
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
	Size	1EG		2EG		4EG		8EG	
WebSphere MQ enabled	1K	251	30	423	60	700	98	**704**	100
(Figure 3 in doc)	4K	131	30						
	16K	44	30						
	64K	12	30						
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
Non WebSphere MQ enabled	Size	1EG		2EG		4EG		8EG	
(Figure 4 in doc)	1K	383	20	628	39	948	75	**1276**	97
	4K	232	20						
	16K	97	20						
	64K	30	20						
Issue an HTTP Request									
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
Request to HTTPValidation	Size	1EG		2EG		4EG		8EG	
(Figure 5 in doc)	1K	190	10	359	19	620	37	**972**	70
	4K	113	10						
	16K	43	10						
	64K	11	10						
High Volume									
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
HTTPInput node to HTTPReply node	Size	1EG		2EG		4EG		8EG	
	1K	1142	36	1771	66	**2340**	93	2254	90
	4K	1020	47						
	16K	637	60						
	64K	209	68						
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
HTTPRequest calling HTTPInput node to HTTPReply node	Size	1EG		2EG		4EG		8EG	
	1K	228	12	429	24	732	45	**1112**	80
	4K	145	12						
	16K	55	12						
	64K	15	12						

Solaris

Figure 10. Throughput results on Solaris

Façade Back End		Scaling for 1K Message Only							
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
	Size	1EG		2EG		4EG		8EG	
WebSphere MQ enabled	1K	410	57	592	94	**602**	100	581	100
(Figure 3 in doc)	4K	237	58						
	16K	83	57						
	64K	22	57						
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
Non WebSphere MQ enabled	Size	1EG		2EG		4EG		8EG	
(Figure 4 in doc)	1K	675	37	1003	69	**1207**	100	1153	100
	4K	442	37						
	16K	186	37						
	64K	57	37						
Issue an HTTP Request									
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
Request to HTTPValidation	Size	1EG		2EG		4EG		8EG	
(Figure 5 in doc)	1K	229	17	429	30	721	50	**1075**	85
	4K	147	17						
	16K	60	17						
	64K	17	15						
High Volume									
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
HTTPInput node to HTTPReply node	Size	1EG		2EG		4EG		8EG	
	1K	1746	57	2575	90	**2670**	100	2618	100
	4K	1450	63						
	16K	750	86						
	64K	269	100						
		Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy	Msgs P/S	CPU Busy
HTTPRequest calling HTTPInput node to HTTPReply node	Size	1EG		2EG		4EG		8EG	
	1K	301	20	535	36	902	67	**1139**	92
	4K	207	20						
	16K	89	20						
	64K	26	20						

As you can see on Windows and Solaris, both the HTTPInput Node/HTTP Reply Node pair and the HTTPRequest node perform well. All of the test cases also scale well up to the point where no CPU resources are left, which occurs at different points depending on the test case and the number of CPUs in the machine.

Machine details

Windows

Hardware:

- 1 IBM xSeries 360 with 4 2.00 GHz Intel Xeon processors
- 3 69 GB SCSI hard drives formatted with NTFS
- 4 GB RAM
- 1 GB Ethernet card

Software:

- Microsoft Windows 2000 with Service Pack 4
- WebSphere MQ V6
- WebSphere Message Broker with V6 with Fix Pack 1
- DB2 for Windows V8.1 with Fix Pack 4

Solaris

Hardware:

- 1 Sun Fire V1280 Server with 8 1.2 GHz processors
- 2 72 GB SCSI hard drives
- 1 StorEdge Fast 3510 array with 2 LUN's and fastwrite cache -- 2 275 GB and 16 GB of RAM
- 1 GB Ethernet card

Software:

- Solaris 9
- WebSphere MQ V6
- WebSphere Message Broker V6 with Fix Pack 1
- DB2 for Solaris V8.1 with Fix Pack 4

Conclusion

This article has explained the role and function of the HTTP transport nodes in WebSphere Message Broker V6.0.0.1. It has shown how you can use the nodes to address a number of scenarios, including:

- Facade an existing application, either WebSphere MQ enabled or non WebSphere MQ enabled
- Facade and enrich an existing HTTP service
- Request an HTTP service from within an existing messaging backbone

WebSphere Message Broker HTTP transport nodes let you develop solutions to the above scenarios quickly and easily. Complex recoding of existing services is not needed -- instead they can be HTTP-enabled through drag and drop development of Message Broker flows, thus greatly reduce the cost of incorporating HTTP services into existing enterprise applications. This article also discussed performance and scalability considerations when designing your message flow applications.

Related topics

- WebSphere Message Broker information center
  A single Eclipse-based Web portal to all WebSphere Message Broker V6 documentation, with conceptual, task, and reference information on installing, configuring, and using your WebSphere Message Broker environment.
- WebSphere Message Broker documentation library
  WebSphere Message Broker specifications and manuals.
- WebSphere Message Broker product page
  Product descriptions, product news, training information, support information, and more.
- Web services support in WebSphere Message Broker V6
  From the WebSphere Message Broker V6 Information Center.
- Working with HTTP flows in WebSphere Message Broker V6
  From the WebSphere Message Broker V6 Information Center.
- WebSphere Message Broker V6 performance reports
  These IBM SupportPacs profile the performance characteristics of WebSphere Message Broker V6 on various platforms, including message rates when using a variety of product functions.
- Performance Harness for Java Message Service (JMS)
  A useful tool to test and drive any Message Broker flow including those with HTTP nodes.
- WebSphere Message Broker samples and Toolkit Samples Gallery
  Shows you how to use HTTP nodes in a facade of an existing WebSphere MQ enabled application.
- Most popular WebSphere trial downloads
  No-charge trial downloads for key WebSphere products.