This documentation is the confidential and proprietary information of Sabre Inc. Any unauthorized use, reproduction, preparation of derivative works, performance, or display of this document, or software represented by this document, without the express written permission of Sabre Inc., is strictly prohibited.

Sabre, Sabre Travel Network, Sabre Airline Solutions and Sabre APIs are trademarks and/or service marks of an affiliate of Sabre. All other trademarks, service marks, and trade names are the property of their respective owners.

Disclaimer of Warranty and Limitation of Liability

This software and any compiled programs created using this software are furnished “as is” without warranty of any kind, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. No oral or written information or advice given by Sabre, its agents or employees shall create a warranty or in any way increase the scope of this warranty and you may not rely on any such information or advice.

Sabre does not warrant, guarantee, or make any representations regarding the use, or the results of the use, of this software, compiled programs created using this software, or written materials in terms of correctness, accuracy, reliability, currentness, or otherwise. The entire risk as to the results and performance of this software and any compiled applications created using this software is assumed by you. Neither Sabre nor anyone else who has been involved in the creation, production or delivery of this software shall be liable for any direct, indirect, consequential, or incidental damages (including damages for loss of business profits, business interruption, loss of business information, and the like) arising out of the use of or inability to use such product even if Sabre has been advised of the possibility of such damages.
# Table of Contents

Chapter 1: ........................................................................................................................................... 4
Persistent Connection Overview .................................................................................................................. 4
  Sabre APIs Persistent Connection URLs ............................................................................................... 5
  Persistent Connection Pooling Versus Sabre APIs Session Pooling ...................................................... 6
Chapter 2: .................................................................................................................................................. 7
Java-Based Persistent Connections ............................................................................................................ 7
  Axis 1 .................................................................................................................................................... 7
  Axis 2 .................................................................................................................................................... 8
  CXF ..................................................................................................................................................... 9
  Spring-WS ........................................................................................................................................... 9
  Configuring Commons HTTP Client ...................................................................................................... 10
  Homegrown ......................................................................................................................................... 11
Chapter 3: ................................................................................................................................................ 12
.NET-based Persistent Connections .......................................................................................................... 12
  .NET Framework ............................................................................................................................... 12
  Homegrown ....................................................................................................................................... 12
Chapter 4: ................................................................................................................................................ 13
Persistent Connections – A Generic Approach ....................................................................................... 13
Chapter 5: ................................................................................................................................................ 15
High Availability, Failover, and Load Balancing ...................................................................................... 15
Persistent Connection Overview

The geographical location of client’s data center affects observed *Sabre APIs* response times. The further away from Tulsa, OK (where *Sabre APIs* are hosted) the higher the response times are. The difference in response times that geographically distributed clients observe is caused by network infrastructure. The longer the distance that data packets need to travel the longer it takes for the packets to reach their destination. The time needed to transfer data from one server to another is called network latency. Both request and response are subject to network delays. The sum of request and response delays is called round-trip latency.

Creating an https connection can be a particularly time consuming operation. The client and the server need to perform a TCP/IP handshake and an SSL handshake. The TCP/IP handshake needs 2 network roundtrips and the SSL handshake requires another 2 roundtrips. The total of 4 roundtrip waits gets reduced to 3 since the second TCP/IP roundtrip and the first SSL handshake exchange happen in parallel. The diagram below shows how an https connection is initiated.
Each roundtrip is subject to network latency. The total time it takes to create an https connection is 3 times network round-trip latency. Let’s take for example a data center located in Munich, Germany. Ping from the data center to Tulsa shows 140ms on average. 140ms times 3 gives 420 ms. 420 ms is the time needed to setup the https connection before any application data exchange can be initiated.

This document discusses how to use persistent connections eliminate the need to continuously re-establish connections to Sabre APIs. Persistent connections are long lived TCP/IP connections that can handle multiple https requests during their life span. The idea is to maintain a pool of opened connections and use them as needed. Depending on the location, clients may see up to 1 second reduction in response times per Sabre APIs invocation thanks to persistent connections.
**Sabre APIs** Persistent Connection URLs

The following URLs are where persistence is currently enabled/supported:

<table>
<thead>
<tr>
<th>Environment</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sabre APIs certification with CERT Backend</td>
<td><a href="https://sws-crt.cert.havail.sabre.com">https://sws-crt.cert.havail.sabre.com</a></td>
</tr>
<tr>
<td>Sabre APIs production Backend</td>
<td><a href="https://webservices.havail.sabre.com">https://webservices.havail.sabre.com</a></td>
</tr>
<tr>
<td>Sabre APIs certification with CERT Backend (Airline Solutions customers)</td>
<td><a href="https://sws-crt-as.cert.havail.sabre.com">https://sws-crt-as.cert.havail.sabre.com</a></td>
</tr>
<tr>
<td>Sabre APIs production Backend (Airline Solutions customers)</td>
<td><a href="https://webservices-as.havail.sabre.com">https://webservices-as.havail.sabre.com</a></td>
</tr>
</tbody>
</table>

**Persistent Connection Pooling Versus Sabre APIs Session Pooling**

Persistent connection pooling operates on the network layer. It minimizes the overhead of TCP/IP and SSL handshakes. Sabre APIs session pooling operates on the application layer. It reduces the overhead of creating Sabre APIs sessions for each invocation. It’s possible that a single TCP/IP connection is used to handle calls with different Sabre APIs sessions. Likewise, subsequent Sabre APIs calls with the same session may be handled by different TCP/IP connections. There is no need to synchronize these two pools.

It’s always recommended to pool sessions. On the other hand, it’s recommended to use a persistent connections pool if it produces significant response time gains.
Chapter 2:

Java-Based Persistent Connections

The implementation of persistent connections in Java differs depending on the library that is used to invoke Sabre APIs. This chapter discusses how to enable persistent connections for the most popular libraries.

**Axis 1**

Axis natively supports persistent connections. To enable persistent connections in Axis:

- Add [Jakarta Commons HTTP Client](http://jakarta.apache.org/commons/httpclient) to your classpath.
- Configure Axis to use CommonsHTTPSender instead of the default HTTPSender implementation. This can be done by adding the following client-config.wsdd file to the client application classpath:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="commonsHTTPConfig"
    xmlns="http://xml.apache.org/axis/wsdd/
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <!-- use CommonsHTTPSender instead of the default HTTPSender -->
    <transport name="http" pivot="java:org.apache.axis.transport.http.CommonsHTTPSender" />
    <transport name="local" pivot="java:org.apache.axis.transport.local.LocalSender" />
    <transport name="java" pivot="java:org.apache.axis.transport.java.JavaSender" />
</deployment>
```

You should tune the following settings that define how persistent connections are handled:

- “Total Connections” Pool size – total number of persistent connections in the pool. This number should be large enough to accommodate expected max
transactions per second to \textit{Sabre APIs} that your application instance must handle

\begin{verbatim}
AxisProperties.setProperty(
    DefaultCommonsHttpClientProperties.MAXIMUM_TOTAL_CONNECTIONSPROPERTY
    _KEY,
    maxTotalConnections);
\end{verbatim}

- “Connections per host” pool size – usually this setting should be equal to “Total Connections”. In case your application uses persistent connections with Axis to systems other than \textit{Sabre APIs} “Connections per host” should be equal to expected max transactions per second that your application instance must handle and “Total Connections” should be equal to max number of connections that your application will need while communicating with all the systems where persistent connections are enabled.

\begin{verbatim}
// "Connections per host" pool size
AxisProperties.setProperty(
    DefaultCommonsHttpClientProperties.MAXIMUM_CONNECTIONS_PER_HOST_PROPERTY
    _KEY,
    maxConnectionsPerHost);
\end{verbatim}

- “Connection pool timeout” – defines how many milliseconds the application should wait for a connection lease. It may happen that all connections in the pool are busy servicing request. When another request comes in your application will wait for a connection to be available. The right setting for this timeout is application specific. The setting should not be too low as your application will not be able to get a connection from a busy pool. It should not be too high as it will cause the application to appear as unresponsive.

\begin{verbatim}
// max duration to wait for a connection from the pool
AxisProperties.setProperty(
    DefaultCommonsHttpClientProperties.CONNECTION_POOL_TIMEOUT_KEY,
    connectionPoolTimeout);
\end{verbatim}

\textbf{Axis 2}

Axis 2 by default use Commons HTTP client. To configure the client you need to create a connection manager object. You then have to pass the connection manager object to the http client constructor and inject the client to Axis 2 configuration context:

\begin{verbatim}
MultiThreadedHttpConnectionManager conmgr = new
    MultiThreadedHttpConnectionManager();
conmgr.getParams().setDefaultMaxConnectionsPerHost(10);
    HttpClient client = new HttpClient(conmgr);
    configurationContext.setProperty(HTTPConstants.CACHED_HTTP_CLIENT, client);
\end{verbatim}

See Configuring Commons HTTP Client for more information about configuring
Commons HTTP Client for persistent connections.

**CXF**

CXF client currently relies on Java HTTP client to send requests. The default Java HTTP Client has limited support for persistent connections. If you use JDK 6 persistent connections are enabled by default. The following properties define the behavior:

```properties
http.keepAlive=<boolean>
default: true

 Indicates if keep alive (persistent) connections should be supported.

http.maxConnections=<int>
default: 5
```

Indicates the maximum number of connections per destination to be kept alive at any given time.

There have been [discussions](#) in the CXF community to allow plugin Commons HTTP Client to CXF. The Commons client offers better control over persistent connections. Currently, it’s possible by implementing a custom conduit. Several people have reported a successful implementation but no source code is available.

To gain more control over persistent connections we suggest using the generic approach as described in the section of this document titled, “Persistent Connections – A Generic Approach.”

**Spring-WS**

Spring-WS by default uses Java HTTP client. The client has limited functionality when it comes to connection pooling. If you use JDK 6 persistent connections are enabled by default. The following properties define the bahviour:

```properties
http.keepAlive=<boolean>
default: true

 Indicates if keep alive (persistent) connections should be supported.

http.maxConnections=<int>
default: 5
```

Indicates the maximum number of connections per destination to be kept alive at any given time.

Spring-WS can be switched to Jakarta Commons HttpClient to send HTTP requests. This is done by passing CommonsHttpServletRequest to WebServiceTemplate as shown in the
example below:

```xml
<bean id="webServiceTemplate"
  class="org.springframework.ws.client.core.WebServiceTemplate">
  <constructor-arg ref="messageFactory"/>
  <property name="messageSender">
    <bean class="org.springframework.ws.transport.http.CommonsHttpMessageSender">
      <property name="credentials">
        <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
          <constructor-arg value="john"/>
          <constructor-arg value="secret"/>
        </bean>
      </property>
    </bean>
  </property>
  <property name="defaultUri" value="http://example.com/WebService"/>
</bean>
```

CommonsHttpMessageSender allows to inject a Jakarta HttpClient. See Configuring Commons HTTP Client for more information about configuring Commons HTTP Client for persistent connections.

**Configuring Commons HTTP Client**

Spring-WS and Axis 2 rely on Jakarta Commons HTTP client to send HTTP requests. The client can maintain persistent connections and pool them as needed. This section describes how to configure the client.

HttpClient can be configured by passing an instance of MultiThreadedHttpConnectionManager to the constructor. MultiThreadedHttpConnectionManager accepts a set of configuration values via getParams().setParams(HttpConnectionManagerParams) call. HttpConnectionManagerParams has the following noteworthy settings (excerpt from the documentation):

- `setDefaultMaxConnectionsPerHost(int maxHostConnections)` - *Sets the default maximum number of connections allowed for a given host config.* The default (2) is low. We recommend setting this to a higher value.

- `setMaxConnectionsPerHost(HostConfiguration hostConfiguration, int maxHostConnections)` - *Sets the maximum number of connections to be used for the given host config.* The default (2) is low. We recommend setting this to a higher value.

- `setMaxTotalConnections(int maxTotalConnections)` - *Sets the maximum number of connections allowed. The default (20) may be too low. We recommend you investigate connection patterns and determine the right value for your application.*

- `setSoTimeout(int timeout)` - *Sets the default socket timeout*
(SO_TIMEOUT) in milliseconds which is the timeout for waiting for data. A timeout value of zero is interpreted as an infinite timeout. We recommend defining a read timeout so that the application remains responsive even if there is a connectivity issue.

- setConnectionTimeout(int timeout) - Sets the timeout until a connection is established. A value of zero means the timeout is not used. The default value is zero. We recommend defining a read timeout so that the application remains responsive even if there is a connectivity issue.

- setStaleCheckingEnabled(boolean value) - Defines whether stale connection check is to be used. Disabling stale connection check may result in slight performance improvement at the risk of getting an I/O error when executing a request over a connection that has been closed at the server side. We recommend enabling the checks to limit impact on the application should connectivity issues occur.

### Homegrown

Please refer to the section of this document titled, “Persistent Connections – A Generic Approach.”
Chapter 3: .NET-based Persistent Connections

.NET Framework

.NET framework supports persistent connections by default. You just need to configure your application to point to the persistent connections URL’s as described the section of this document titled, “Sabre APIs Persistent Connection URLs.”

Homegrown

Please refer to the section of this document titled, “Persistent Connections – A Generic Approach.”
Persistent Connections – A Generic Approach

The generic approach to persistent connections is to add a proxy to your infrastructure. A proxy is a web server that is responsible for forwarding http(s) requests on behalf of the application. Your application connects to the proxy and the proxy forwards the request to the server. Many proxies have the capability to maintain persistent connections. The application doesn’t have to change. It still uses transient connections but this time it points to the proxy. The proxy accepts transient connections and forwards traffic using a pool of persistent connections that it maintains.

The rest of the section describes how to setup a proxy using Apache HTTPD, a popular and free web server.

The following procedure explains the process of setting up an apache based proxy for sws-sts environment:

1. Install Apache with mod_proxy and mod_proxy_http:
   http://httpd.apache.org/docs/2.2/install.html
2. Configure the proxy by adding the following lines to your httpd.conf

   # load appropriate modules in httpd.conf LoadModule
   proxy_module modules/mod_proxy.so LoadModule
   proxy_http_module modules/mod_proxy_http.so

   # configure the proxy
   # the following example takes requests for all urls (“/”) and
   # forwards them as is to https://sws3-crt.cert.sabre.com/
   # pool of 5 persistent connections will be created upon first
   # request
   # if volumes require, the pool will grow up to 100 connections
   # keepalive=on tells Apache to send KEEP_ALIVE packets — that
   # prevents firewalls from dropping connections
   # this directive works for a proxy that accepts http
ProxyPass / https://sws3-crt.cert.sabre.com/ min=5 max=100 keepalive=on

If you would like to set a proxy that accepts https place the following config in your ssl virtual host:
SSLProxyEngine on
ProxyPass / https://sws3-crt.cert.sabre.com/ min=5 max=100 keepalive=on

4. Point your application to the proxy. You should be able to observe transient connections to the proxy but persistent sockets from the proxy to Sabre APIs.

As you can see the proxy mode is not very different from standard Apache configuration. There are just a few commands that need to be added to Apache configuration. We need to load proxy and http_proxy modules, map local address to the remote address with ProxyPass. Refer to ProxyPass directive documentation for a full list of options.
High Availability, Failover, and Load Balancing

The previous chapter discussed how to inject a proxy between your application and Sabre APIs. The proxy accepts transient connections and maintains persistent connections to Sabre APIs. The setup allows us to reduce response times without changing the application. This chapter discusses how to deploy proxies to achieve high availability.

We recommend two approaches for deploying persistent connections proxies:

1. **Centralized** – a farm of identical proxy instances is installed behind a hardware load balancer. This option is recommended to companies that already have hardware load balancers in their datacenters and have expertise to operate them.

2. **Distributed** – each application instance gets its own proxy instance. This option is recommended for data centers that currently don’t use hardware load balancers.

**Centralized Deployment**

The diagram below shows the centralized deployment. Application servers talk to a hardware load balancer. Typically, hardware load balancers are deployed in active/passive pairs to avoid single node of failure. The load balancer forwards requests to proxy instances. Proxies maintain persistent connections to Sabre APIs.

Connections from application servers to hardware load balancers do not have to be persistent. Since the application servers and load balancers are physically located close to each other the time lag for creating a TCP/IP connection is negligible. Same applies to connections from the hardware load balancer to the proxies. Even if these connections are transient, it will not adversely impact response times. To reduce the response time only wide area network connections need to be persistent.
In this deployment model the hardware load balancer talks to a pool of proxies. The load balancer also monitors the proxies. If a proxy is down it will be disabled and no requests will be forward to it. Load balancers also perform load balancing. Load balancing policies can vary between simple round robin and sophisticated algorithms were resource consumption on the proxy servers determines how much traffic they get. Thanks to this behavior application servers connecting to a single load balanced virtual IP can be sure that their requests will be serviced as long as there is at least one instance of the proxy running. The application servers do not have to maintain a list of available proxies as the load balancer masquerades the proxy farm.

Note on security: the diagram below indicates that traffic to and from load balancer can be either HTTP or HTTPS. The choice of the protocol depends on the deployment environment and sensitivity of the data transmitted. The choice of the protocol (http vs. https) in local area networks doesn’t significantly change response times.

**Distributed Deployment**

The diagram below shows the distributed deployment. In this deployment each application instance has its own proxy instance. When a proxy goes down the associated application should also be taken out of traffic. Although the proxy is a separate process on the server it’s considered as a part of the application. If the proxy goes down the associated application instance should also be taken out of traffic.

Typical resource requirements of a proxy process are negligible compared to resources needed by application. Therefore, deploying proxies with the application should not cause immediate capacity concerns.

Since each application instance talks only to its own proxy instance load balancing is achieved by distributing traffic across application instances. The same applies to failover. With this deployment model the proxy benefits from the same load balancing and failover mechanism that the application itself uses.
Application
Server #1
HTTPS

Application server #2
HTTPS

Application server #n

Internet

Sabre APIs