



OSP Toolkit

Internal Architecture

Release 3.1.2

1 July 2004

Revision History

Revision	Date of Issue	Changes
2.7.0	March 12 th , 2003	Added revision history.
2.8.0	March 20 th , 2003	No Changes
2.8.1	March 25 th , 2003	No Changes
2.8.2	April 9 th , 2003	No Changes
2.9.0	June 1 st , 2003	No Changes
2.9.1	July 7 th , 2003	No Changes
2.9.2	July 28 th , 2003	Modified the Introduction to include the Interoperability documents.
2.9.3	Sep 15 th , 2003	No changes
2.11.1	Feb 12 th , 2004	No changes
3.0	March 11 th , 2004	No changes
3.1	April 8 th , 2004	Updated the document with changes for load balancing.
3.1.1	May 12 th , 2004	No Changes
3.1.2	July 1 st , 2004	No Changes

Contents

Revision History	2
Contents	3
Introduction	4
Major Components	5
Provider	5
Transaction.....	5
Message Info.....	7
Message Queue	7
Message Exchange	7
Communications Manager	7
SSL Session.....	8
HTTP Connection	8
Message Formatting and Parsing	10
XML Tree.....	11
XML Document	12
SMIME Signature.....	12
MIME Block	12
Token Information	12
Utility Services.....	12
Statistics Accumulator	13
Delay Probe	14
Transaction ID Tracking	14
Load Balancing in the toolkit	15

E-mail: support@transnexus.com

www.transnexus.com

Copyright © 2003 by TransNexus. All Rights Reserved.

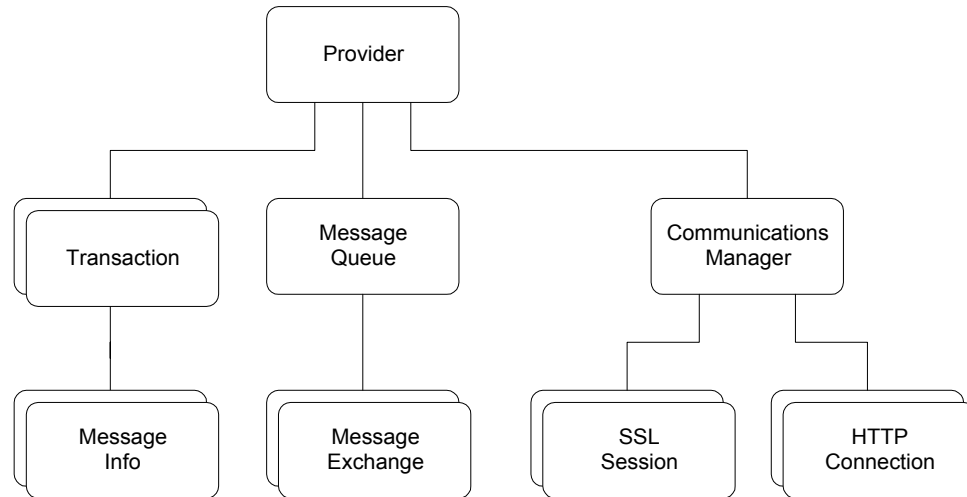
This document describes the internal architecture of release 3.1.2 of the Open Settlement Protocol (OSP) Toolkit. That Toolkit, freely available under license from TransNexus, contains an implementation of the standard settlement protocol endorsed by the European Telecommunications Standards Institute (ETSI) and the International Multimedia Teleconferencing Consortium's Voice over IP (VoIP) Forum. The Toolkit also implements, as an option, extensions to the standard that allow access to enhanced services.

The OSP Toolkit contains fourteen separate documents, including this one. The documents are:

- *Introduction*
- *H.323 Implementation Guide*
- *SIP Implementation Guide*
- *How to Build and Test the OSP Toolkit*
- *Error code List*
- *Programming Interface*
- *Cisco Interoperability Example*
- *Device Enrollment*
- *Internal Architecture*
- *Porting Guide*
- *SIP – OSP Interoperability Test Cases*
- *H.323 – OSP Interoperability Test Cases*
- *Protocol Extensions*
- *ETSI Technical Specification TS 101 321*

The *OSP Toolkit Introduction* includes a “Document Roadmap” section that summarizes the various documents and their application. The sections that follow describe the Toolkit's major components, its formatting and parsing functions, high-level utility services, and platform-specific services. Although implemented entirely in ANSI C, the Toolkit relies on an object-oriented architecture. Consequently, the sections of this document are based on the major objects that comprise that architecture.

Note that this document is not intended as an exhaustive description of all the objects in the Toolkit library, including details of all their members and member functions. It serves, instead, as an orientation to the design of the library. To avoid any problems that might arise from out-of-date or faulty documentation, library objects are fully documented in the actual source code.



• Figure 1 Major Toolkit Software Objects.

Major Components

At a high level, nine major components perform the Toolkit's essential services. Figure 1 shows those major components, and it illustrates the relationships between them. The following subsections describe each component in greater detail.

Note:

Figure 1 does not represent a complete or rigid object hierarchy for the Toolkit library. Rather, it simply shows the major components of the library and their relationship with each other. Other objects, described in later sections of this document, are also part of the library.

Provider

The provider object is the parent object for all other aspects of the Toolkit library. It contains all the information pertaining to a particular settlement service provider, and its public interface is defined by the Toolkit Programming Interface document.

The major child objects of the provider object are transaction objects, a message queue, and a communication manager. Other, less significant, child objects are also part of the provider object; they are described in the "Utility Services" section below.

Transaction

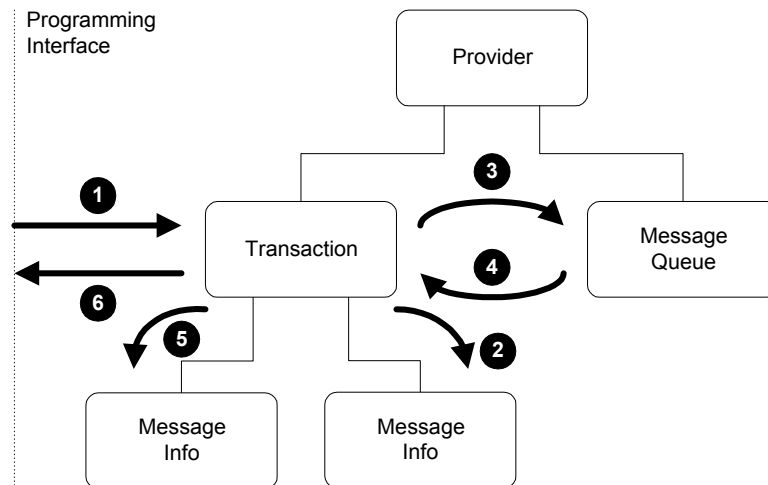
The Toolkit uses a transaction object to represent a single transaction (e.g. a phone call) with a settlement provider. Transaction objects are children of a provider object, and multiple transactions may be active simultaneously with the same parent provider. The public interface of a transaction object is defined in the Toolkit Programming Interface document.

The main functions of a transaction object are creating OSP messages and interpreting responses received from settlement servers. The transaction object uses message info objects to represent these messages.

Processing within transaction object member functions takes place in the thread of the application that calls the Toolkit interface. Figure 2 shows the general flow of execution within the transaction object. All of the steps in that figure execute in the calling program's thread.

The steps in the figure are:

- 1) An application makes a call to the Toolkit programming interface and invokes a transaction object member function.
- 2) If the call requires that the Toolkit send a message to the settlement service provider, the transaction object creates a message info object for that message.
- 3) The transaction object converts the message info object to a message exchange object and adds the message exchange to the provider's message queue and waits for a reply. The message exchange object, described below, includes a mechanism that allows the transaction object to block while waiting for that reply. The actual transmission of the message and the reception of a reply take place in a separate thread controlled by the provider's communication manager.
- 4) When the reply is received, the transaction object thread is reactivated and it retrieves the reply from the message exchange object.
- 5) To interpret the reply, the transaction object parses the reply into a message info object.
- 6) The transaction object then returns control to the calling application, returning information contained in the reply.



• Figure 2 Flow of Execution within a Transaction Object

Message Info

The message info object represents the information content of an OSP message. This form is not the raw bytes that are transmitted or received, but rather a more abstract representation of the message.

Message Queue

The message queue object stores OSP messages waiting to be transmitted to a service provider, and it serves as the bridge between transaction objects and the provider object's communication manager. As those two objects execute in different threads, the message queue also provides thread synchronization services. Most of that synchronization mechanism is implemented in the message exchange objects, but the message queue does include a condition variable that can wake up the communication manager when the queue transitions from empty to non-empty.

Message Exchange

Message exchange objects represent an OSP message and its associated reply (or error code). They contain raw OSP messages destined for settlement servers and the replies received from those servers. They also include a condition variable to synchronize different threads that may access the exchange information. In the case of message exchange objects on the provider's message queue, the condition variable synchronizes the thread in which the transaction object runs with threads controlled by the provider's communication manager. Thread synchronization takes place as follows:

- 1) When a transaction object creates an OSP message, it puts that message in a message exchange object and adds the object to the message queue. The transaction thread then blocks on the message's condition variable.
- 2) The communication manager retrieves the message from the queue and transmits it to the settlement server. When a reply arrives, the communications manager adds the reply to the message exchange object and changes the value of the condition variable. It can also modify the condition variable if an error occurs.
- 3) The transaction object thread wakes up when the condition variable changes and processes the reply or handles the error, as appropriate.

Communications Manager

The communications manager object coordinates communications to and from the settlement provider server. It does so by managing pools of Secure Sockets Layer (SSL) sessions and HTTP connections. Its typical operation proceeds as follows.

- 1) The communications manager waits for the provider's message queue to change from empty to non-empty, indicating that a message is ready for transmission.
- 2) When a message arrives on the queue, the communication manager identifies an HTTP connection on which to send the message. In looking for the connection, the communication manager uses the following priority:

If the number of current connections is less than the maximum permitted for the provider, create a new HTTP connection object.

If the number of connections is at the configured limit, select one of the current connections and transfer the message. In selecting a connection, the manager employs a round robin scheme to look for an existing HTTP connection object that is currently idle. This is a part of the load balancing algorithm implemented in the toolkit. If there are no idle connections, wait for a busy connection to become idle.

3) If the communication manager had to create a new HTTP connection object for the message, and if SSL security is enabled with the provider, it also assigns an SSL session to that connection. SSL sessions are assigned as follows:

a) Look for an existing SSL session object that is not currently assigned to an HTTP connection object.

If no unassigned SSL session objects exist, create a new SSL session object.

Once the message is assigned to an SSL session and HTTP connection, the communications manager has completed its responsibilities with respect to the message.

SSL Session

An SSL session object maintains the information needed to create or resume secure socket layer sessions. That information includes the session identifier and the associated symmetric encryption key.

SSL session objects are created by the communication manager in conjunction with the creation of a new HTTP connection object. SSL session objects, however, are not destroyed with the destruction of the associated HTTP connection. Instead, the communication manager keeps SSL sessions in a pool for re-assignment to a new HTTP connection. The communication manager only destroys SSL objects when the provider itself is destroyed.

SSL session objects do include a lifetime that limits the validity of the data they contain. Logically, this lifetime can expire either while the session is assigned to an HTTP connection, or while the session is inactive and waiting for re-assignment. In the actual implementation, HTTP connection objects are responsible for verifying the session lifetime prior to sending each message. If that lifetime has expired (or if the settlement server requests a new SSL session), the information in the SSL session object is updated to reflect the new SSL session information.

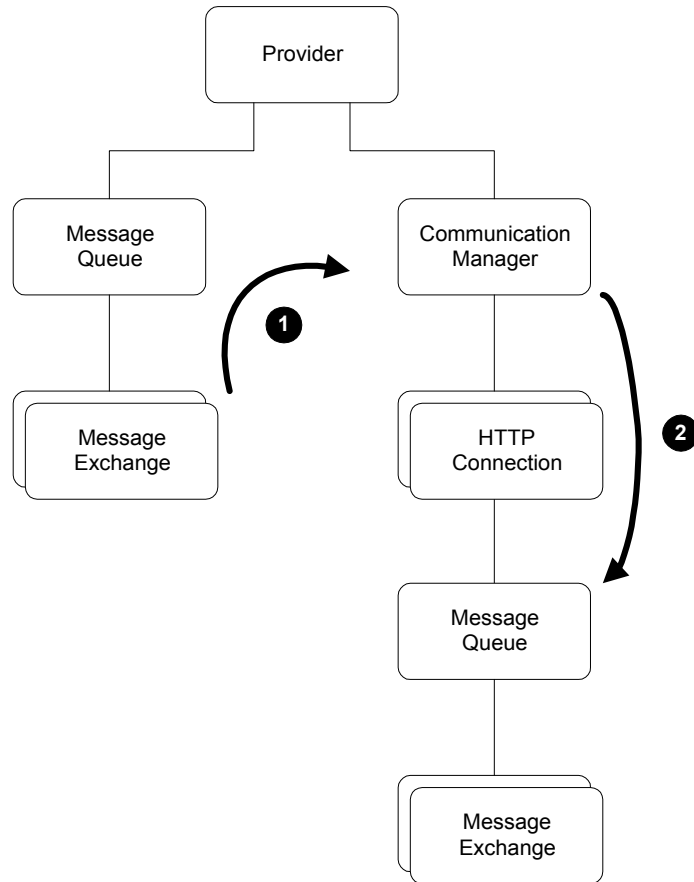
HTTP Connection

HTTP connection objects control a Hypertext Transfer Protocol connection to a settlement server. Their major function is managing the network input and output for that connection, and handling the (minimal) HTTP protocol processing for the connection. Each connection object contains its own message queue that holds messages pending transmission. The communications manager object puts message exchange objects on this queue. As Figure 3 indicates, the typical scenario is for the communication manager to remove

messages from the provider's message queue, assign them to a specific HTTP connection object, and add the message to that connection's queue.

Each HTTP connection object executes in its own thread, and it is synchronized to the connection manager by a condition variable in its message queue. On creation, the connection object immediately waits for this condition variable to indicate that a message is on the queue. When a message appears, the connection object takes the following steps to transmit the message to the settlement server:

- 1)** The connection object attempts to open a TCP connection with the settlement server. Each connection object contains its own list of servers. The list for each connection is in different order (rotated by one). It begins with the first server in the list, and, should that attempt fail, continues with subsequent servers on the list until a connection is established or the list is exhausted.
- 2)** If the connection object cannot open a connection, it sleeps for a specified period of time and tries again (returning to step 1). The sleep time and maximum number of retries are controlled by configuration parameters.
- 3)** Once the connection is established, the connection object checks the lifetime of its SSL session. If that lifetime is still valid, the connection object attempts to resume the SSL session. If the session has expired, however, (or if the server rejects the session), the connection object renegotiates the SSL session parameters and updates the SSL session object.
- 4)** The connection object then transmits the message (with appropriate HTTP headers) to the settlement server and blocks, waiting for a reply.
- 5)** If a connection error occurs before the reply arrives, the connection object restarts the process at step 1.
- 6)** When the connection object receives the reply, it ties it to the original message exchange object, removes that message exchange from its message queue, and signals the original transaction object through the message exchange's condition variable.



• Figure 3 Message Flow to HTTP Connection Objects.

- 7) The connection object then checks its message queue for additional messages, if any are present, it processes them beginning with step 3.
- 8) If the message queue is empty, the HTTP connection object's thread blocks on its message queue condition variable, waiting for the communications manager to add new messages to the queue. The connection object includes a maximum time to be blocked that is equal to the TCP persistence time configured for the provider.
- 9) If a new message arrives before the persistence time expires, the connection object begins processing at step 3.
- 10) If no message arrives, the connection object returns its SSL session object to the communication manager's pool and terminates its thread.

Message Formatting and Parsing

In addition to managing the communications between applications and settlement service providers, another significant role of the Toolkit software is message formatting and parsing. As the contents of this section indicate, Toolkit includes support for several standards: XML, S/MIME, MIME, Base64, and ASN.1 DER.

The process of creating a formatted message for transmission proceeds in five stages, beginning with a message info object. As Figure 4 illustrates, the message is modified or augmented at each stage.

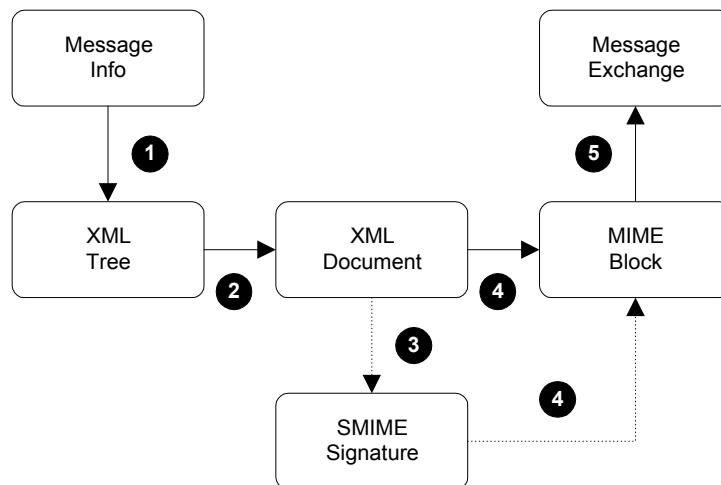
The complementary process, parsing a received message for interpretation, uses the same objects, though in reverse order. In addition, the parsing process may require an additional steps to interpret authorization tokens. Figure 5 (on the following page) illustrates the entire process. The additional step required for tokens is base64 decoding.

The following subsections describe the objects of the formatting and parsing processes. The starting and ending points—message info and message exchange objects—were discussed in the preceding section.

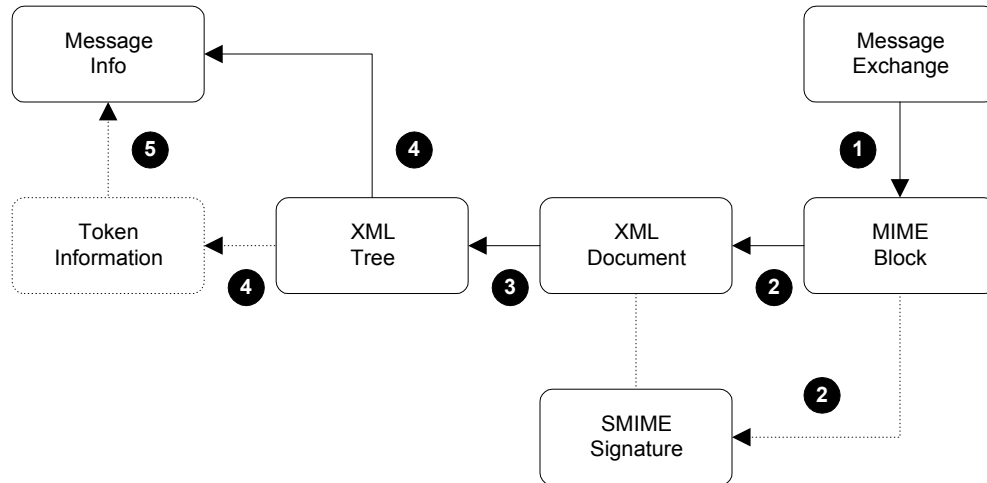
XML Tree

The XML tree object contains a logical representation of the contents of an XML document. It mirrors the nested logical structure of XML content through tree-like data structures. Each node in the tree represents an XML element; children of a node are sub-elements of the parent. The node also includes the attributes associated with the element.

XML trees may be created in two ways. The formatting process creates an XML tree from a message object. Since message info objects are specific to each OSP message type, this process is itself specific to individual message types. In the reverse direction, XML trees are created from XML documents. This process is a straightforward example of standard XML parsing, though some significant simplifications from true XML parsing are possible. In particular, OSP messages do not use external entity references or processing instructions, so handling of external entities or processing instructions is not required. In addition, the Toolkit library does not validate XML documents, nor does it exhaustively check for well-formedness. (The Toolkit's XML parser is an adherent to the Internet protocol philosophy: be conservative in what you send and liberal in what you accept.)



• Figure 4 Formatting an OSP Message.



• Figure 5 Parsing an OSP Message.

XML Document

The XML document object contains the linear, character string for an XML document. It represents the bit-by-bit content of OSP messages in a format ready for transmission, or as received. For outgoing messages, XML documents are built from XML trees. And for incoming messages, XML documents are extracted from MIME blocks.

SMIME Signature

SMIME signature objects contain a digital signature conforming to the Secure Multipurpose Internet Mail Extension standards. During formatting, SMIME signature objects are generated directly by the cryptographic services. For received messages, SMIME signatures are extracted from MIME blocks.

MIME Block

MIME block objects contain a complete OSP protocol message (which is, by definition, a MIME block transferred by HTTP). When messages are prepared for transmission, MIME blocks are formed by combining an XML document with an SMIME signature. In the reverse direction, MIME blocks are simply extracted from message exchange objects.

Token Information

OSP authorization response messages may include tokens to validate the authorization. These tokens, if present, are base64 encoded in the XML content. Before passing the token to an application, it must be decoded. The also Toolkit includes functionality to interpret the contents of an authorization token. (That functionality is needed to verify tokens, for example.)

Utility Services

The Toolkit includes several utility objects in that support the Open Settlement Protocol implementation. The major utility objects, each described in this section, include a delay

probe, a statistics accumulator, and a transaction ID tracking object. Figure 6 shows the relationship of these objects with the Toolkit's primary objects.

Statistics Accumulator

The statistics accumulator object is used to track network performance statistics for a call. Specifically, it can keep track of either one-way or round trip delay measurements as the application reports them. Each transaction object includes two statistics accumulators, one for each measure quantity.

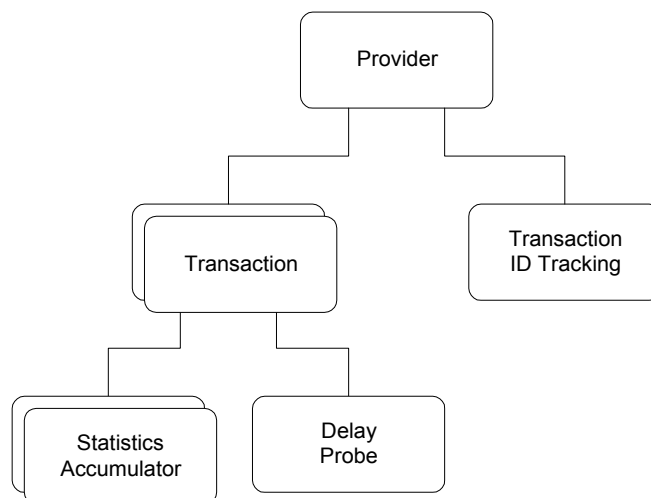
Each statistics accumulator includes the following parameters:

m	minimum measured value
n	number of measurements
\bar{x}	sample mean
q	sum of square of samples
S^2	sample variance

As the application collects sample measurements, it reports those measurements to the Toolkit using the following values:

\hat{m}	minimum measured value within sample set
\hat{n}	number of measurements within sample set
$\hat{\bar{x}}$	mean of sample set
\hat{S}^2	sample variance of sample set

The new sample set is used to update the old parameters (time i) and calculate new parameters (time $i+1$) according to the following equations:



• Figure 6 Utility Service Objects.

$$\begin{aligned}m_{i+1} &= \min(m_i, \hat{m}) \\n_{i+1} &= n_i + \hat{n} \\ \bar{x}_{i+1} &= \frac{n_i \cdot \bar{x}_i + \hat{n} \cdot \hat{x}}{n_{i+1}} \\ q_{i+1} &= q_i + (\hat{n} - 1) \cdot \hat{S}^2 + \hat{n} \cdot \hat{x}^2 \\ S_{i+1}^2 &= \frac{q_{i+1} - n_{i+1} \cdot \bar{x}_{i+1}^2}{n_{i+1} - 1}\end{aligned}$$

Delay Probe

The delay probe object is responsible for performing a round-trip latency test to one or more destinations. Such tests may be used to support quality of service extensions to the basic OSP standard. The delay probe performs its function by opening sockets and sending small UDP datagrams to the echo service of each destination. It then measures the time until a response is received. The delay probe incorporates a maximum time to wait for responses.

Transaction ID Tracking

The transaction ID tracking object ensures the uniqueness of transaction IDs included in authorization tokens. It is designed to help prevent the inappropriate re-use of those tokens, and it makes sure that the same transaction ID is not re-used within the lifetime of the authorization token that contains it.

Load Balancing in the toolkit

The toolkit can connect with multiple Service Points at the same time and distribute load across them. Load Balancing in the toolkit is implemented by

1. Employing a round robin mechanism in the communication manager to select a connection object. A round robin scheme is a simple way of ensuring that consecutive messages are handed over to different connection objects.
2. Having a list of Service Points in each connection object rather than all the connection objects using the same list. The list for each connection object is rotated by one position. This ensures that connection objects connect to a different SP.
3. Limiting the number of messages on each connection. This ensures that we send only a specific number of messages to each service point and then try to move on to the other service points configured.

Every time the communication manager has a message to transmit, it goes through the following algorithm to select a connection object. It then hands over the message to the connection object selected.

- If the current number of connection objects is less than the maximum allowed, create a new connection object. Randomize the list of SP's in the communication manager, and copy it to the connection object.
- If it cannot create any new connection objects, select a connection on the round-robin basis that is currently idle.
- If there are no idle connections, wait for a busy connection object to become idle.

When a message appears on the connection object queue, the connection object takes the following steps to transmit the message to the server:

- The connection object checks to see if it is already connected to a server. If yes, {then it transmits the request}.
- If the connection object is not connected to a server, then it checks to see if the object was ever connected with any of the servers. (The cases are (1) that the connection object has just been started and it has not yet connected with any server, and (2) the connection was made but it either expired or deleted). If the connection was never made to any of the servers in the list, pickup the first server in the list and try connecting. If the connection was made then pickup the next server in the list and try connecting.
- The failure and retry cases are handled based on the retries configured. Every time a failure occurs, the current message count within the connection object is reset.
- If the message is sent successfully, it increments the current message count in the connection objects. It then checks to see if the current message count = the maximum messages allowed on the connection. If Yes, it deletes the connection by disconnecting from this server. Also, it resets the current message count.

Since the connection objects fail over from one service point (SP) to another, there is a chance that the connection objects converge to just one SP in the list, and all of them start

loading that particular SP. To avoid this problem we restrict the maximum number of messages that can be sent over a connection with each SP. After this limit has been exceeded, the connection object tears down its current connection and moves on to the next SP in its list. This allows connection objects to reconnect with the SP that was down for a brief period of time.

Configuring *ospvmessagecount* for different servers under different situations

Let us consider that the user wants to configure Service Points and Connection Objects under these conditions:

- 1. Achieve equal load balancing among the SP's:** In this case, the user should configure an equal value of *ospvmessagecount* for all the SP's. If each of the server can do 10 calls/sec, and if the user wants a connection to persist for at least 30 secs, then the user should set the *ospvmessagecount* parameter to 300 (30*10) for each of the SP. If the specifications are not clear, the user can set this value to 1000. In order to achieve equal load balancing, the number of CO's should be a multiple of the number of service points configured. For instance, if there are 3 SP's configured and the number of CO's is set to 9, then 3 CO's would lock to each SP and we can achieve an equal load (as long as all the SP's are up and running. Under fluctuating conditions it may not be easy with this algorithm to ensure equal load).
- 2. Achieve 2:1 load balancing between the set {SP0,SP1} and {SP2}:** In this case (given the 10 calls/sec requirement above), the user should configure *ospvmessagecount* for SP0 and SP1 to 600, and for SP2 to 300. This would make the connections to SP0 an SP1 to last twice as long as the connection to SP2. Assuming 6 CO's, 2 CO would be locked to each SP. Since the *ospvmessagecount* for SP0 and SP1 is twice that of SP2, the number of messages going to SP0 and SP1 would be twice as many as that going to SP2.
- 3. Have SP0, SP2, and SP4 as ACTIVE and SP1, SP3, and SP5 as STANDBY:** This requires the toolkit to send messages to SP1, SP3, and SP5 only when SP0, SP2, and SP4 are not available. To achieve this, the sequence of SP configured (using one of the 3 API mentioned above) should be – SP0, SP1, SP2, SP3, SP4, and SP5 (notice that we have active and standby servers alternately in the list). The *messagecount* for SP0, SP2 and SP4 should be a high number - lets say 10,000, which means that a very high number of messages (10,000) can be sent to these SP over any single connection. The *messagecount* for SP2 should be fairly small (lets say 5), so that the COs do not send a lot of messages to backup service points and try to lock onto either the active service points quickly. The reason for having the sequence as SP0, SP1, SP2, SP3, SP4, and SP5 is that we want to configure a Standby server between 2 Active Servers. This enables the COs that had originally connected to a Backup server to subsequently move on and connect to a different active server. Thus, if there were 6 CO, 3 would immediately lock to Active servers. The other 3 would initially connect to the Backups, and then move on to connect to SP0, SP2, and SP4. Now, if SP0 went down, the CO locked to SP0 would eventually move to SP2 and lock there. However, since the message count for each SP is set to a finite number, there will be a time when the message count on SP5 equals 10,000, and then the CO locked to SP5 would move to SP0 and lock to it (provided SP0 is up now). This way we can protect the CO's from locking on to a Active server infinitely and never trying the other Active server that had gone down temporarily.

Idle Connection Object Selection mechanism

The current algorithm used for selecting an idle connection is extremely CPU intensive. Currently, the selection algorithm runs 'while' loop to check if there is an idle connection. And it continues to run the loop, in iteration, as long as it does not find an idle connection.

To improve the selection algorithm, we will introduce a condition variable that will wait on a connection object to become idle and not run the loop iteratively. This would improve the CPU utilization and enhance performance.

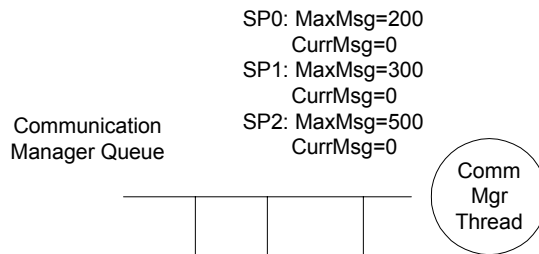
Potential Issue

This algorithm might potentially result in a Ripple/Wave effect where all the CO's lock to one SP at any given time and then switch over to the next one almost at the same time. This could happen if one of the 2 SP's configured is down and all the CO's lock on to the currently "UP" SP at the same time. Then, they would all exceed their maximum message limit almost at the same time and switch to the next. This see-saw phenomenon will continue.

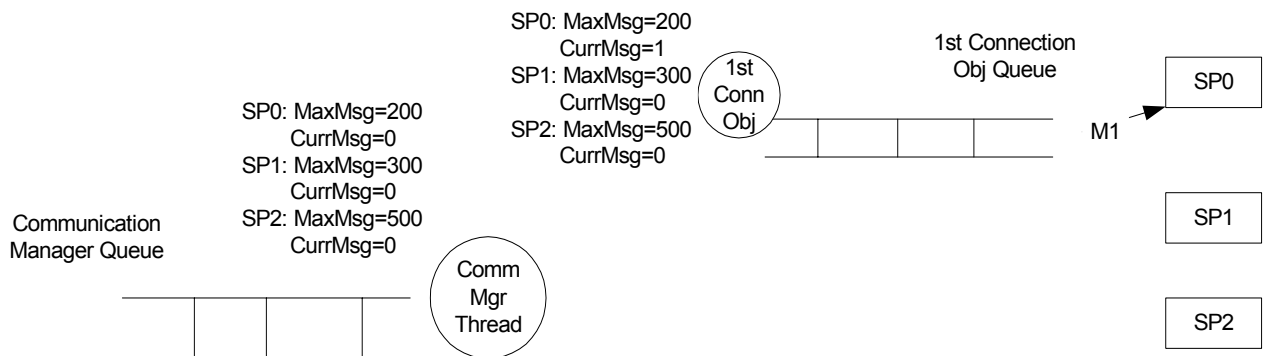
Example

Lets take an example. There are 3 SP's: SP0, SP1, SP2 with message counts of 200, 300, and 500 respectively. The maximum number of http connection objects that can be created is 3. There are 1000 messages to be sent. Service Points SP1 and SP2 crash when the message M4 is passed to the toolkit. SP1 and SP2 come back when the 150th and 200th message is being sent. The sequence of events will be as drawn below.

1. OSPPProviderNew is called.



The communication manager and its queue is created. The SP's and their message per connection capacity is defined within the manager.

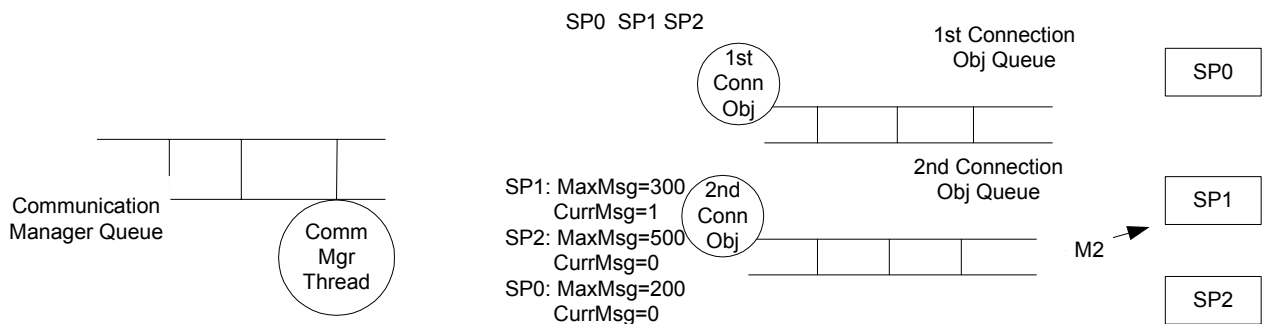


- The 1st OSPPTTransactionAuthorizationRequest API call is made.

On receiving the message, the communication manager checks if the current number of connection objects is less than the maximum number configured in ProviderNew. Since at this time there are no connection objects, the manager launches a new connection obj and passes it the message – M1. The list of service points is also copied into the connection object. Since this is the first connection object, the sequence of SP's is the same as that in the communication manager. After this step, the CurrMsg count in the connection object is set to 1.

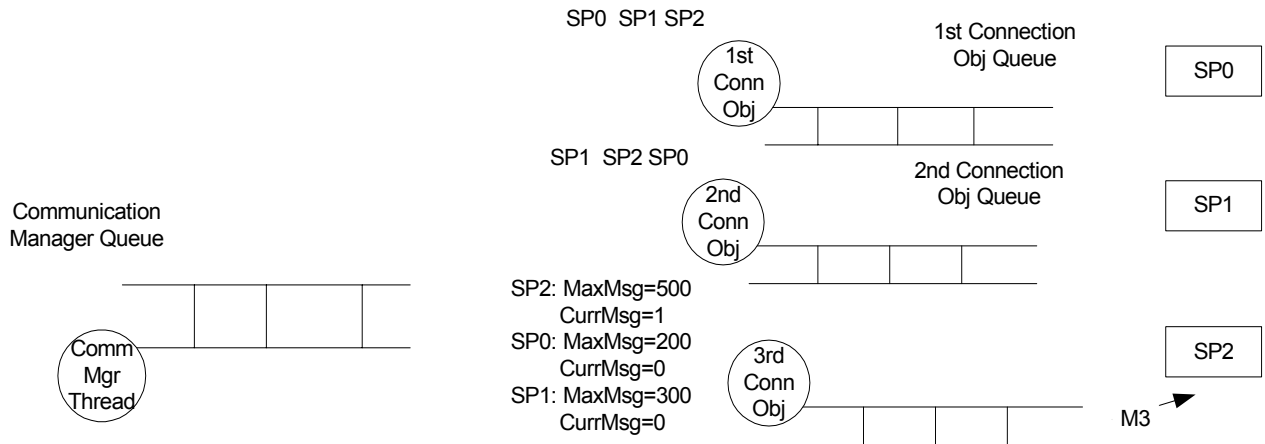
- The 2nd OSPPTTransactionAuthorizationRequest API call is made.

On receiving the message, the communication manager again checks if the current number of connection objects is less than the maximum number configured in ProviderNew. Since at this time there is just one connection object, the manager launches another connection obj and passes it the message – M2. The list of service points is also copied into the connection object. The sequence of SP's is now not the same as that in the communication manager. After this step, the CurrMsg count for SP1 is set to 1.



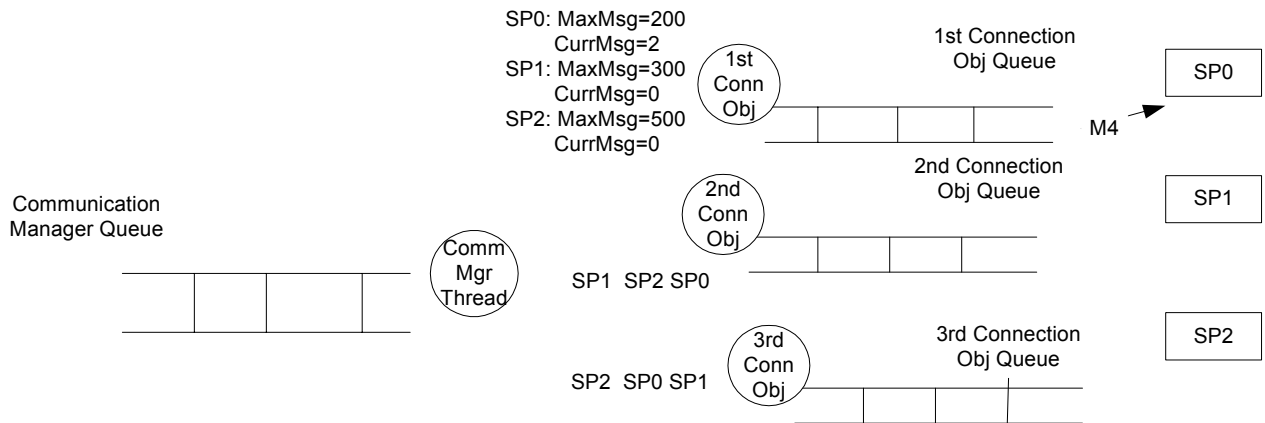
- The 3rd OSPPTTransactionAuthorizationRequest API call is made.

On receiving the message, the communication manager again checks if the current number of connection objects is less than the maximum number configured in ProviderNew. Since at this time there are two connection objects, the manager launches another connection obj and passes it the message – M3. The list of service points is also copied into the connection object. The sequence of SP's is now not the same as that in the communication manager or 2nd Conn object. After this step, the CurrMsg count for SP2 is set to 1.



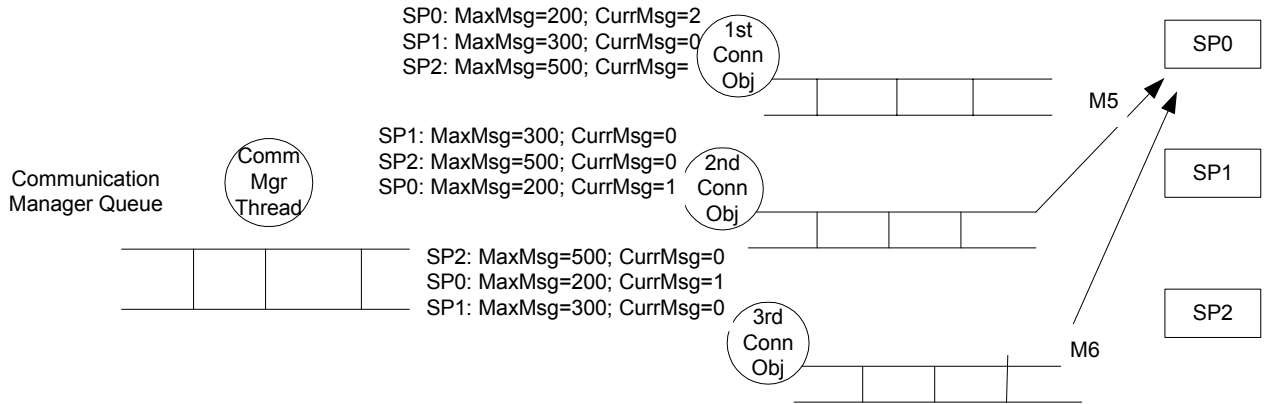
5. Message M4 needs to be sent.

Now that the manager cannot create any more connection objects, it starts a round robin algorithm to pass messages on to one of the 3 connection object queues. It thus decides to pass it on to the 1st connection object.

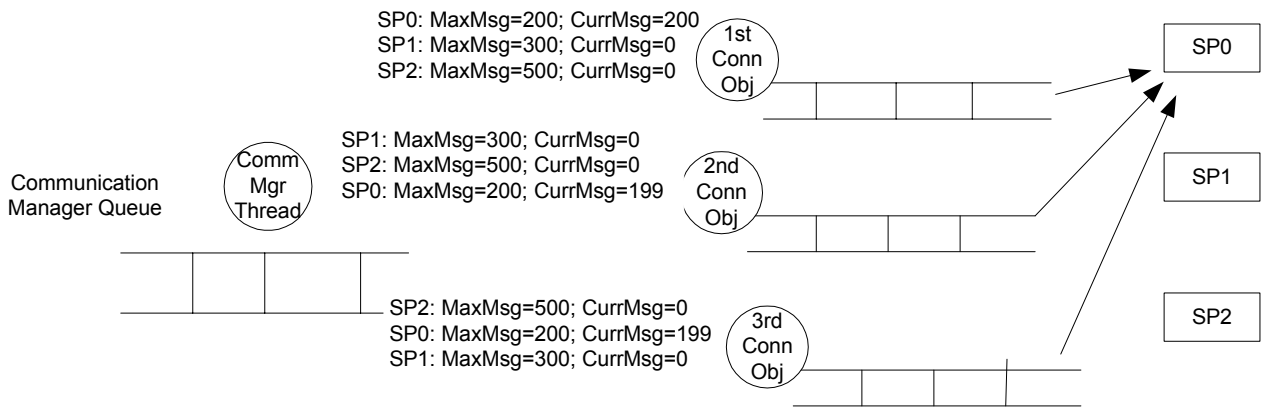


6. Message M5 and M6 needs to be sent.

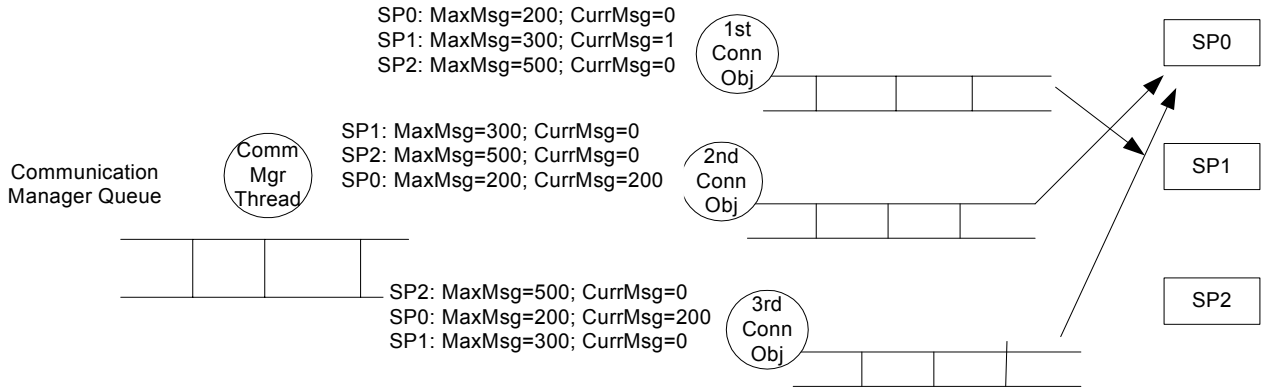
Based on the round robin algorithm, the communication manager sends the messages M5 and M6 to 2nd and 3rd object respectively. However, since SP1 and SP2 are down, the 2nd connection obj traverses through its list only to lock itself at SP0. Similarly 3rd connection obj also locks itself at SP0. Thus, both 2nd and 3rd CO reset their previous message counts, and set the current message count for SP0 to



- The next 594 messages get equally distributed between the 3 CO (198 to each). Each of these messages ends up at SP0. Although SP1 and SP2 recover while these messages are being sent to SP0, none of the connection objects move away from SP0 because they get locked to it. However, after 594 messages, the message count for CO1 reaches 200, and it switches from SP0 to SP1.



- For the next three messages, 2 of these go to SP0 (through CO2 and CO3, which are still locked on to SP0). The other one goes to SP1 (through CO1).



- The remaining 597 messages go to SP1 as all the CO's now lock to SP1.

