

OPEN SIGCOMP™

Interface Description



About this Document

This document describes the Application Programmers Interface to the Open SigComp™ stack. “Open SigComp” is a trademark of Estacado Systems, LLC.

Contact Information

For further information on the content of this document, please contact:

Adam Roach
214-329-0491
adam@estacado.net

Document History

Version	Date	Author	Comments
0.1	12/14/2005	Adam Roach	First Version
0.2	04/21/2006	Adam Roach	Updates from Review

Table of Contents

1	Introduction.....	1
1.1	Interface Model.....	1
1.2	Overview of Operation	2
1.2.1	Stack Construction.....	2
1.2.2	Compression	2
1.2.3	Decompression.....	3
1.2.4	Compartment Maintenance.....	3
2	API Objects.....	4
2.1	Type Reference	4
2.2	osc::Compressor Class Reference.....	5
2.3	osc::DeflateCompressor Class Reference.....	6
2.4	osc::SigcompMessage Class Reference.....	7
2.5	osc::Stack Class Reference	8
2.6	osc::StateChanges Class Reference	11
2.7	osc::StateHandler Class Reference.....	12
2.8	osc::TcpStream Class Reference.....	13
3	Examples.....	14
3.1	Stack Construction.....	14
3.2	UDP.....	14
3.2.1	Encoding.....	14
3.2.2	Decoding.....	14
3.2.3	Zero-Copy Encoding.....	15
3.3	TCP.....	15
3.3.1	Encoding.....	15
3.3.2	Decoding.....	15
4	References.....	17

1 Introduction

The Open SigComp™ stack is Estacado Systems' implementation of the SigComp protocol defined in RFC 3320 [1] and RFC 4077[2]. This document describes the classes and methods that an application would use to interface with the stack.

For information about other aspects of the product, see [5], [6] and [7]. For general information about the SigComp protocol, see RFC 3320 [1] and RFC 4077 [2].

1.1 Interface Model

The model used to interface with the Open SigComp™ stack is very similar to that described in RFC 3320 [1]. The only substantive difference is that the Open SigComp™ stack relegates receiving and sending of network data to the application. The architecture used by the Open SigComp™ stack for interfacing is shown in Figure 1. The components in this diagram are defined and described in RFC 3320 [1].

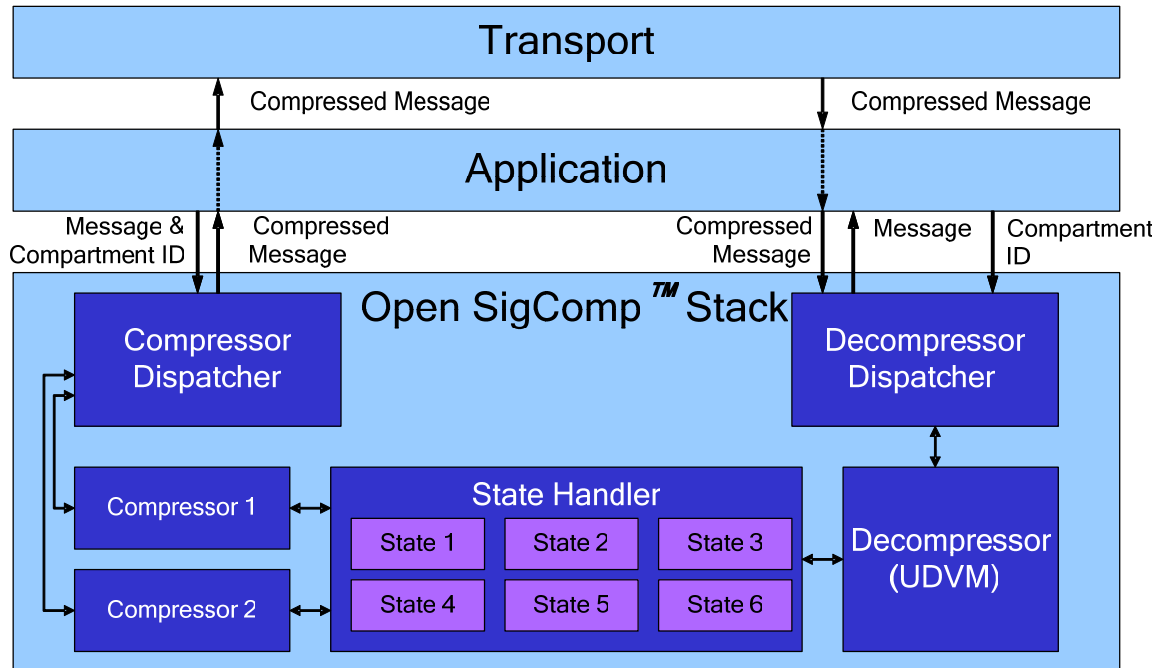


Figure 1: Open SigComp™ Interface Model

One key concept in the SigComp protocol is that of compartments. States created by the Universal Decompressor Virtual Machine (UDVM) reside within compartments, which (among other purposes) serve as quota systems to ensure that remote endpoints cannot create excessive amounts of state.

A compartment maps to an association with a remote endpoint. So, for example, in an environment that uses SigComp between an edge proxy and mobile endpoints, the proxy would maintain a compartment for each remote endpoint with which it is interacting. Similarly, the

mobile endpoints would maintain a compartment for each edge proxy with which it is interacting (typically, one).

To allow maximum flexibility, the Open SigComp™ stack gives the application complete freedom in choosing the format used for Compartment Identifiers – identifiers can even be of varying lengths. The only restriction placed on such identifiers is that they must be comparable using straight bitwise comparisons.

1.2 Overview of Operation

1.2.1 Stack Construction

At the heart of the Open SigComp™ stack is the StateHandler. It keeps track of the following:

- all information relating to remote endpoint capabilities
- states relating to previously decompressed messages
- information necessary to correlate NACK messages to the proper compartment so that corrective action can be taken.

The StateHandler uses synchronization to ensure multithread safeness.

The primary application interface of the Open SigComp™ stack is the Stack object. For performance reasons, the Stack object is not synchronized; each instance of a Stack can be accessed from only one thread at a time. If multiple threads need to compress and/or decompress SigComp messages, they must each have their own instance of a Stack.

As a consequence of the foregoing, applications must create a StateHandler object. They then create one or more Stack objects, as is appropriate for their threading model, providing each stack a reference to the StateHandler instance.

Additionally, before any messages can be compressed, the application must create one or more Compressor objects and add them to the Stack object(s). In the current version of the Open SigComp™ stack, the only defined Compressor object is the DeflateCompressor.

Once the application has created a StateHandler and one or more Stack objects (each with at least one Compressor), the Open SigComp™ stack is ready for use.

1.2.2 Compression

To compress a message, the application calls `compressMessage` on the Stack instance. The application must provide the data to be compressed along with a compartment identifier.

The `compressMessage` method returns a `SigcompMessage` instance, from which a buffer can be extracted for sending to the network.

Because of framing considerations, the method used to extract a buffer from a SigComp message varies depending on whether the compressed message is to be sent over a stream-oriented transport (e.g. TCP) or a datagram-oriented transport (e.g. UDP).

1.2.3 Decompression

The mechanism used to decompress a message received from the network varies depending on whether the message is received over a stream-oriented transport (e.g. TCP) or a datagram-oriented transport (e.g. UDP).

When using a stream-based transport, the application must create a `TcpStream` instance for each open connection. Any data received from the stream-based transport is placed into its corresponding `TcpStream` buffer, using the `addData` method. The application then calls `uncompressMessage` on the `Stack` with the `TcpStream` instance repeatedly, until the `uncompressMessage` call returns a length of 0.

When using a datagram-based transport, the application simply calls `uncompressMessage` on the `stack` for each received datagram.

If the most recent call to `uncompressMessage` returns 0, the application should check whether a NACK message needs to be sent to indicate an error in decompression. It performs this check by calling the `getNack` method on the `Stack`. If this call returns a `SigcompMessage` object, then the application must serialize and send the `SigcompMessage` to the source of the most recent received message.

Once a message has been successfully decompressed, the application should verify that the decompressed data constitutes a valid message. If the resulting message is acceptable, the application must call `provideCompartmentId` on the `Stack`, so that the states created by the message decompression get stored for future use.

1.2.4 Compartment Maintenance

Because there are often circumstances in which the association may go away between endpoints without explicit signaling, the application must periodically instruct the `StateHandler` to clean up compartments that have not been used for some period of time. The precise amount of time after which a compartment is considered “stale” is left up to the application developer.

On a periodic basis, the application is responsible for calling the `removeStaleCompartments` method on the `StateHandler` instance. Doing so first checks all compartments for a flag that indicates that they are “old.” Any such compartments are deleted, and states that they contain are subject to deletion. This method then marks all remaining compartments as “old.” The next time the compartment is used, the “old” flag is reset. Subsequently, if a compartment is not used between the times that the `removeStaleCompartments` method is called, it will be destroyed.

Such clean-up should generally be performed rather infrequently. For example, for many applications, calling `removeStaleCompartments` once every 24 hours should provide adequate results.

2 API Objects

This section includes information for the classes that are used directly by the application. Only the methods that are to be used by the application are described in this document.

Any methods present on the classes other than those documented in this section are subject to change in future versions of the Open SigComp™ stack.

2.1 Type Reference

The Open SigComp stack uses a number of custom data types that are exposed to the application. These types are described below.

osc::byte_t

Used to represent a byte, typically for buffer purposes. This is likely to be aliased to type “unsigned char” on most platforms.

osc::count_t

Used to designate the number of objects in a collection. This will be an unsigned integer of undetermined length.

osc::u16

Used to represent an unsigned 16-bit integer. Equivalent to the ISO C99 type “uint16_t”.

osc::u32

Used to represent an unsigned 32-bit integer. Equivalent to the ISO C99 type “uint32_t”.

osc::u8

Used to represent an unsigned 8-bit integer. Equivalent to the ISO C99 type “uint8_t”.

2.2 `osc::Compressor` Class Reference

```
#include <Compressor.h>
```

Pure virtual base class for all compressors. The application does not need to access any methods of the `Compressor` class.

It is possible that descendants of the `Compressor` class may contain methods for modifying the operation of the compressor themselves. The current implementation does not contain any classes for which this is the case.

`osc::Compressor::Compressor (osc::StateHandler & sh)`

Constructor for `osc::Compressor`.

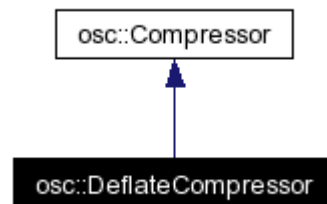
Parameters:

stateHandler Reference to state handler that this compressor uses to store and retrieve compression-related state. This must be the same `StateHandler` that is used by the stack to which the compressor is to be added.

2.3 *osc::DeflateCompressor* Class Reference

```
#include <DeflateCompressor.h>
```

Inheritance diagram for *osc::DeflateCompressor*:



The *DeflateCompressor* class implements a SigComp compressor based on the Deflate algorithm described in RFC 1951[4]. The application does not need to access any methods of the *DeflateCompressor* class except for the constructor.

***osc::DeflateCompressor::DeflateCompressor* (*osc::StateHandler* & *stateHandler*)**

Constructor for *osc::DeflateCompressor*.

Parameters:

stateHandler Reference to state handler that this compressor uses to store and retrieve compression-related state. This must be the same *StateHandler* that is used by the stack to which the compressor is to be added.

2.4 `osc::SigcompMessage` Class Reference

```
#include <SigcompMessage.h>
```

This class encapsulates a SigComp message. From an API perspective, it is used only for serialization of messages for transmission to the network.

SigcompMessage instances are serialized differently depending on whether they are to be sent using a datagram oriented protocol (e.g. UDP) or using a stream oriented protocol (e.g. TCP).

`size_t osc::SigcompMessage::getDatagramLength () const`

Returns the length of the buffer returned by `getDatagramMessage`.

See also:

`getDatagramMessage` (p.7)

`osc::byte_t * osc::SigcompMessage::getDatagramMessage ()`

Returns a pointer to a buffer appropriate for sending to the network over a datagram-oriented transport.

The returned buffer remains the property of the **`SigcompMessage`** (p.7) instance, and is valid as long as the **`SigcompMessage`** (p.7) is not mutated.

See also:

`getDatagramLength` (p.7)

`size_t osc::SigcompMessage::getStreamLength ()`

Returns the length of the buffer returned by `getStreamMessage`.

See also:

`getStreamMessage` (p.7)

`osc::byte_t * osc::SigcompMessage::getStreamMessage ()`

Returns a pointer to a buffer appropriate for sending to the network over a stream-oriented transport.

The returned buffer remains the property of the **`SigcompMessage`** (p.7) instance, and is valid as long as the **`SigcompMessage`** (p.7) is not mutated.

See also:

`getStreamLength` (p.7)

2.5 *osc::Stack* Class Reference

```
#include <Stack.h>
```

The Stack is the main interface between the application and the Open SigComp™ Stack. In general, the application creates one StateHandler object globally. It then creates a Stack object for each thread that will be compressing and/or decompressing messages.

***osc::Stack::Stack* (*osc::StateHandler* & *stateHandler*)**

Constructor for *osc::Stack*.

Parameters:

stateHandler Reference to state handler that this stack uses to store and retrieve compression-related state. A single state handler may be shared among several Stacks.

***void osc::Stack::addCompressor* (*osc::Compressor* * *compressor*)**

Adds the specified compressor to the stack.

Compressors are invoked in the order in which they are added. The first compressor that can create valid output for the message is used for the message. In almost all cases, the first compressor added will be able to compress the message.

It is not possible to remove compressors from the stack once they have been added.

After the application adds a compressor to the stack, the application must not reference the compressor again. If multiple stacks are constructed, each must have its own compressor instance(s).

Parameters:

compressor Pointer to the compressor to be added. The stack takes ownership of the compressor.

***template<typename T> osc::SigcompMessage* * *osc::Stack::compressMessage* (*osc::byte_t* * *m*, *size_t* *s*, *const T* & *id*, *bool* *r* = *false*)**

Convenience method to compress a message when compartment IDs are native types.

Note:

It must be valid to perform byte-wise comparisons on the passed in IDs for this method to be valid.

***osc::SigcompMessage* * *osc::Stack::compressMessage* (*osc::byte_t* * *input*, *size_t* *inputSize*, *const void* * *id*, *size_t* *idLength*, *bool* *reliableTransport* = *false*)**

Compresses an application message into a SigComp message.

Parameters:

input Pointer to a buffer containing the uncompressed application-level message that is to be compressed.

inputSize Number of bytes in the input message

id Pointer to the compartment identifier for the compartment with which this message is to be associated. **Compartment** identifiers are chosen unilaterally by the application.

Open SigComp™

Interface Description

idLength Size of the compartment identifier, in bytes.

reliableTransport Flag indicating whether the request is to be sent over a reliable transport.

Returns:

A **SigcompMessage** (p.7) instance from which an appropriate buffer can be extracted for sending to the network. The application owns this object and is responsible for destroying it.

See also:

SigcompMessage (p.7)

osc::SigcompMessage * osc::Stack::getNack ()

Creates a SigComp NACK message if the previous attempt to decompress a message failed.

Returns:

A pointer to a SigComp message which the application is to send to the address from which the most recent SigComp message was received. The application owns this object and is responsible for destroying it.

Return values:

0 No error has occurred, so no NACK message has been generated.

See also:

RFC 4077 [2]

template<typename T> void osc::Stack::provideCompartmentId (osc::StateChanges * sc, const T & id)

Convenience method to provide compartment ID when compartment IDs are native types.

Note:

It must be valid to perform byte-wise comparisons on the passed in IDs for this method to be valid.

void osc::Stack::provideCompartmentId (osc::StateChanges * stateChanges, const void * id, size_t idLength)

Provides a compartment id for a decompressed message.

Parameters:

stateChanges Pointer to the **StateChanges** (p.11) object that was handed to the application as part of the uncompressMessage operation. The **Stack** (p.8) takes ownership of this object.

id Pointer to the compartment identifier for the compartment with which the **StateChanges** (p.11) are to be associated. Compartment identifiers are chosen unilaterally by the application.

idLength Size of the compartment identifier, in bytes.

size_t osc::Stack::uncompressMessage (osc::TcpStream &, osc::byte_t * output, size_t outputSize, osc::StateChanges *&)

Decompresses a stream-oriented SigComp message.

Parameters:

- tcpStream* Reference to a **TcpStream** (p.13) object from which the compressed messages are to be extracted.
- output* Pointer to a buffer into which the uncompressed application message is to be placed.
- outputSize* Number of bytes available in output buffer. The amount of output generated will never exceed 65535 bytes in size.
- stateChanges* Reference to a pointer into which the stack will write a pointer to a new **StateChanges** (p.11) object. The application will use this **StateChanges** (p.11) object to provide a compartment ID once it verifies the uncompressed message. (See provideCompartmentId). If the application does not call provideCompartmentId with this **StateChanges** (p.11) object, then it is responsible for deleting it.

Returns:

Number of bytes written into the output buffer (i.e. size of the decompressed message).

Return values:

- 0 An error has occurred during decompression or there is not sufficient data in the **TcpStream** (p.13) object to comprise a full SigComp message.

size_t osc::Stack::uncompressMessage (osc::byte_t * input, size_t inputSize, osc::byte_t * output, size_t outputSize, osc::StateChanges *& stateChanges)

Decompresses a datagram-oriented SigComp message.

Parameters:

- input* Pointer to a buffer containing a compressed SigComp message.
- inputSize* Number of bytes present in the input buffer.
- output* Pointer to a buffer into which the uncompressed application message is to be placed.
- outputSize* Number of bytes available in output buffer. The amount of output generated will never exceed 65535 bytes in size.
- stateChanges* Reference to a pointer into which the stack will write a pointer to a new **StateChanges** (p.11) object. The application will use this **StateChanges** (p.11) object to provide a compartment ID once it verifies the uncompressed message. (See provideCompartmentId). If the application does not call provideCompartmentId with this **StateChanges** (p.11) object, then it is responsible for deleting it.

Returns:

Number of bytes written into the output buffer (i.e. size of the decompressed message).

Return values:

- 0 An error has occurred during decompression.

2.6 osc::StateChanges Class Reference

```
#include <StateChanges.h>
```

This class represents the state changes requested by a successfully decompressed SigComp message. If the application deems the decompressed message valid, it calls `osc::Stack::provideCompartmentId` with these changes to accept them (and assign them to a compartment).

The application does not create, examine, or mutate `StateChanges` objects. Every method on the `StateChanges` class, including the constructor, is subject to change in future versions of the Open SigComp™ product.

2.7 osc::StateHandler Class Reference

```
#include <StateHandler.h>
```

The **StateHandler** stores state for the UDVM as well as tracking NACK messages and compartments.

It is responsible for the destruction of old states and compartments, which it handles through reference counting. When ever a state or compartment is retrieved from the **StateHandler** it must be released when the retriever is finished using it.

The **StateHandler** also provides a method of removing stale compartments; by calling `removeStaleCompartments` all stale compartments will be discarded from the state handler and all the state that was unique to those compartments will be discarded as well. A stale compartment is defined with in the context of **StateHandler** as a **Compartment** that has neither been retained nor released since the last time `removeStaleCompartments` was called.

NACK to compartment associations are automatically removed by age once a threshold of associations is met. This threshold is computed by multiplying the per-compartment threshold by the number of compartments. Because of this, there is no guaranteed minimum number of associations that will be stored per-compartment.

See also:

RFC 3320 [1] Section 6

```
osc::StateHandler::StateHandler (size_t maxStateSpaceInBytes = 8192, size_t maximumCyclesPerBit = 64, size_t maxDecompressionSpaceInBytes = 8192, osc::u16 sigcompVersion = 0x01, osc::count_t nackThreshold = 4)
```

Creates a **StateHandler** with a defined maximum state size and a maximum cycles per bit.

Parameters:

maxStateSpaceInBytes The `state_memory_size` parameter as defined in RFC 3320 [1].

maximumCyclesPerBit The `cycles_per_bit` parameter as defined in RFC 3320 [1].

maxDecompressionSpaceInBytes The `decompression_memory_size` parameter as defined in RFC 3320 [1].

sigcompVersion The version of SigComp supported. In general, 0x02 if NACKs are to be used; 0x01 otherwise. See RFC 3320 [1] and RFC 4077 [2] for details.

nackThreshold Number of messages to be retained per compartment for the purposes of NACK message correlation. This number represents a maximum average, and is not enforced on a per-compartment basis.

```
void osc::StateHandler::removeStaleCompartments ()
```

This method removes all the compartments which have not been used since the last time this method was called.

```
void osc::StateHandler::useSipDictionary ()
```

This method instructs the state manager to advertise and (where possible) make use of the predefined SIP/SDP dictionary defined in RFC 3485 [3]. This action cannot be undone.

2.8 *osc::TcpStream* Class Reference

```
#include <TcpStream.h>
```

The *TcpStream* class is a rolling buffer that queues the data received from the network for a specific TCP connection. It performs framing and un-escaping for messages received from the network. Because of the nature of stream protocols, the *TcpStream* class may contain multiple and/or fractional SigComp messages. In general, after adding data to a *TcpStream* object, the application should attempt to decompress messages from the *TcpStream* until the call to *osc::Stack::uncompressMessage* returns 0.

***osc::TcpStream::TcpStream* ()**

Constructor for *osc::TcpStream*.

***void osc::TcpStream::addData* (*osc::byte_t* * *data*, *size_t* *size*)**

Adds data that has been read from the network.

Parameters:

data Pointer to new data. The ***TcpStream*** (*p.13*) DOES NOT take ownership of the data.

size Number of bytes in the data buffer.

3 Examples

This section contains sample code that demonstrates how applications interact with the Open SigComp™ stack.

3.1 Stack Construction

```
osc::StateHander *sh = new osc::StateHandler(stateLimit, cyclesPerBit,
                                             udvmMemorySize, 2);

sh->useSipDictionary();

osc::Stack *ss = new osc::Stack(*sh);

osc::Compressor *dc = new osc::DeflateCompressor(*sh);

ss->addCompressor(dc);
```

3.2 UDP

3.2.1 Encoding

```
osc::SigcompMessage *sm;
sm = ss->compressMessage(buffer, bufferLen, id, idLength);

if (sm)
{
    sendto(socket, sm->getDatagramMessage(),
           sm->getDatagramLength(), 0, address, sizeof(address));
}
```

3.2.2 Decoding

```
struct sockaddr address;
socklen_t addressLength;

size_t len1 = recvfrom(socket, buffer1, sizeof(buffer1), 0,
                       &address, &addressLength);

osc::StateChanges *sc;

size_t len2 = ss->uncompressMessage(buffer1, len1,
                                    buffer2, sizeof(buffer2),
                                    sc);

if (len2)
{
    /* Process message and Calculate compartment ID here */

    ss->provideCompartmentId(sc, id, idLength);
}
```

```
else
{
    osc::SigcompMessage *nack = ss->getNack();
    if (nack)
    {
        sendto(socket, nack->getDatagramMessage(),
              nack->getDatagramLength(), 0, address, addressLength);
    }
}
```

3.2.3 Zero-Copy Encoding

The zero-copy encoding interface is not yet implemented. When complete, it will allow retrieval of iovec structures from SigcompMessage instances; these iovec structures will be suitable for use with the Berkeley-sockets-style “sendmsg” function.

```
osc::SigcompMessage *sm;
sm = ss->compressMessage(buffer, bufferLen, id, idLength);

if (sm)
{
    struct msghdr header;
    header.msg_iov    = sm->getVector();
    header.msg_iovlen = sm->getVectorLength();

    /* Set up remainder of message header here */

    sendmsg(socket, header, 0);
}
```

3.3 TCP

3.3.1 Encoding

```
osc::SigcompMessage *sm;
sm = ss->compressMessage(buffer, bufferLen, id, idLength, true);

write(socket, sm->getStreamMessage(), sm->getStreamLength());
```

3.3.2 Decoding

```
osc::TcpStream *stream = new osc::TcpStream();

size_t len1 = read(socket, buffer1, sizeof(buffer1));

stream->addData(buffer1, len1);

osc::StateChanges *sc;

size_t len2;

while ((len2 = ss->uncompressMessage(stream, buffer2,
```

Open SigComp™

Interface Description

```
        sizeof(buffer2), sc)) != 0)
{
    /* Process message and Calculate compartment ID here */
    ss->provideCompartmentId(sc, id, idLength);
}

osc::SigcompMessage *nack = ss->getNack();
if (nack)
{
    write(socket, nack->getDatagramMessage(), nack->getDatagramLength());
}
```

4 References

- [1] Price, R., Bormann, C., Christoffersson, J., Hannu, H., Liu, Z., and J. Rosenberg, “Signaling Compression (SigComp)”, RFC 3320, January 2003.
- [2] Roach, A. B., “A Negative Acknowledgement Mechanism for Signaling Compression”, RFC 4077, May 2005.
- [3] Garcia-Martin, M., et al, “The Session Initiation Protocol (SIP) and Session Description Protocol (SDP) Static Dictionary for Signaling Compression (SigComp)”, RFC 3485, February 2003.
- [4] Deutch, P., “DEFLATE Compressed Data Format Specification version 1.3”, RFC 1951, May 1996.
- [5] Estacado Systems, LLC, “Open SigComp: System Architecture”, version 0.1
- [6] Estacado Systems, LLC, “Open SigComp: Detailed Design”, version 0.1
- [7] Estacado Systems, LLC, “Open SigComp: Library Characteristics”, version 0.1