Network Working Group
Internet-Draft
Expires: August 30, 2002

A. B. Roach
dynamicsoft

March 2002

A Negative Acknowledgement Mechanism for Signalling Compression
draft-roach-sigcomp-nack-01

Status of this Memo

Copyright Notice

Abstract

This document describes a mechanism that allows Signalling Compression (SigComp) implementations to report precise error information upon receipt of a message which cannot be decompressed. This negative feedback can be used by the recipient to make fine-grained adjustments to the compressed message before retransmitting it, allowing for rapid and efficient recovery from error situations.

Table of Contents

1. Introduction

Signalling Compression (see reference [1]), often called "SigComp", defines a protocol for transportation of compressed messages between two network elements. One of the key features of SigComp is the ability of the sending node to request that the receiving node store state objects for later retrieval.

1.1 The Problem

While the "SigComp - Extended Operations" document (reference [2]) defines a mechanism that allows for confirmation of state creation, operational experience with the SigComp protocol has demonstrated that there are still several circumstances in which a sender's view of the shared state differs from the reciever's view. A non-exhaustive list of the circumstances in which such failures may occur are detailed below.

1.1.1 Compartment Disposal

In SigComp, stored states are associated with compartments. Conceptually, the compartments represent one instance of a remote application. These compartments are used to limit the amount of state that each remote application is allowed to store. Compartments are created upon receipt of a valid SigComp message from a remote application. In the current protocol, applcations are expected to signal when they are finished with a compartment so that it can be deleted (by using the S-bit in requested feedback data).

Unfortunately, expecting the applications to be well-behaved is not sufficient to prevent state from piling up. Unexpected client failures, reboots, and loss of connectivity can cause compartments to become "stuck" and never removed. To prevent this situation, it becomes necessary to implement a scheme by which compartments that appear disused may eventually be discarded.

While the preceding facts make such a practice necessary, discarding compartments without explicit signalling can have the unfortunate side effect that active compartments are sometimes discarded. This leads to a different view of state between the server and the client.

1.1.2 Client Restart

The prime motivation behind SigComp was compression of messages to be sent over a radio interface. Consequently, most deployments of SigComp will involve a mobile unit as one of the the endpoints. Such units are not generally highly available. Node restarts (due to e.g. a battery running out) will induce situations in which the network-

based server beleives that the client contains several states that are no longer actually available.

1.1.3 Server Failover

Many applications for which SigComp will be used (e.g., SIP [3]) use DNS SRV records for server lookup. One of the important features of DNS SRV records is the ability to specify multiple servers from which clients will select at random, with probabilities determined by the q-value weighting. The reason for defining this behavior for SRV records is to allow load distribution through a set of equivalent servers, and to permit clients to continue to function even if the server with which they are communicating fails. When using protocols that use SRV for such distribution, the traffic to a failed server is typically sent by the client to an equivalent server that can serve the same purpose. From an application perspective, this new server often appears to be the same endpoint as the failed server, and will consequently resolve to the same compartment.

Although SigComp state can be replicated amongst such a cluster of servers, maintaining integrity of such states requires a two-phase commit process, which adds a great deal of complexity to the server, and can degrade performance significantly.

1.2 The Solution

Although SigComp allows returned SigComp parameters to signal that all states have been lost (by setting "state_memory_size" to 0 for one message in the reverse direction), such an approach provides an incomplete solution to the problem. In addition to wiping out an entire compartment when only one state is corrupt or missing, this approach suffers from the unfortunate behavior that it requires a message in the reverse direction that the remote application will authorize. Unless a lower-layer security mechanism is employed (e.g. TLS), this would typically mean that a compressed application-level message in the reverse direction must be sent before recovery can occur. In many cases, such as SIP-based mobile terminals, such messages may be seldom; in others (pure client/server deployments), they won't ever happen.

The proposed solution to this problem is a simple Negative Acknowledgement (NACK) mechanism which allows the recipient to communicate to the sender that a failure has occured. This NACK contains a reason code that communicates the nature of the failure. For certain types of failures, the NACK will also contain additional details that might be useful in recovering from the failure.

2. Node Behavior

The following sections detail the behavior of nodes sending and receiving SigComp NACKs. The actual format and values are described in section Section 3.

2.1 Normal SigComp Message Transmission

Although normal in all other respects, SigComp implementations that use the NACK mechanism need to calculate and store a SHA-1 hash for each SigComp message that they send. This must be stored in such a way that, given the SHA-1 hash, the implementation is able to locate the compartment with which the sent message was associated. Further, when a reliable transport is being used, the implementation must be able to retrieve the plain-text version of the original message.

2.2 Receiving a "Bad" SigComp Message

When a received SigComp message causes a decompression failure, the recipient forms and sends a SigComp NACK message. This NACK message contains a SHA-1 hash of the received SigComp message that could not be decompressed. It also contains the exact reason decompression failed, as well as any additional details that might assist the NACK recipient to correct any problems. See section Section 3 for more information about formatting the NACK message and its fields.

For a connection-oriented transport, such as TCP, the NACK message is sent back to the originator of the failed message over that same connection.

For a stream-based transport, such as TCP, the standard SigComp delimiter of 0xFFFF is used to terminate the NACK message.

For a connectionless transport, such as UDP, the NACK message is sent back to the originator of the failed message at the port and IP address from which the message was sent. Note that this may or may not be the same port to which the appliation would typically receive messages.

> The behavior specified above is strictly necessary for any generally useful form of a NACK mechanism. In the most general case, when an implementation receives a message that it cannot decompress, it has exactly three useful pieces of information: the contents of the message, an indication of why the message cannot be decoded, and the contents of the compressed message. Note that none of these contain any indication of where the remote applicaition is listening for messages, if it differs from the sending port.

2.3 Receiving a SigComp NACK

The first action taken upon receipt of a NACK is an attempt to find the message to which the NACK corresponds. This search is performed using the 20-byte SHA-1 hash contained in the NACK. Once the matching message is located, further operations are performed based on the compartment that was associated with the sent message.

Further behavior of a node upon receiving a SigComp NACK depends on whether a reliable or unreliable transport is being used.

2.3.1 Unreliable Transport

When SigComp is used over an unreliable transport, the application has no reasonable expectation that the transport layer will deliver any particular message. It then becomes the application layer's responsibility to ensure that data is retransmitted as necessary. In these circumstances, the NACK mechanism relies on such behavior to ensure delivery of the message, and never performs retransmissions on the application's behalf.

When a NACK is received for a message sent over an unreliable transport, the NACK recipient uses the contained information to make appropriate adjustments to the compressor associated with the proper compartment. The exact nature of these adjustments are specific to the compression scheme being used, and will vary from implementation to implementation. The only requirement on these adjustments is that they must have the effect of compensating for the error that has been indicated (e.g. by removing the state that the remote node indicates it cannot retreive).

In particular, when an unreliable transport is used, the original message must not be retransmitted by the SigComp layer upon receipt of a NACK. Instead, the next application initiated transmission of a message will take advantage of the adjustments made as a result of processing the NACK.

2.3.2 Reliable Transport

When a reliable transport is employed, the application makes a basic assumption that any message passed down the stack will be retransmitted as necessary to ensure that the remote node receives it unless a failure is indicated by the transport layer. Because SigComp acts as a shim between the transport-layer and the application, it becomes the responsibility of the SigComp implementation to ensure that any failure to transmit a message is communicated to the application.

When a NACK is received for a message sent over a reliable transport, the SigComp layer must indicate to the application that an error has occured. In general, the application should react in the same way as it does for any other transport layer error, such as a TCP connection reset. For most applications, this reaction will initially be an attempt to reestablish the connection, and re-initiate the failed transaction.

2.4 Detecting Support for NACK

Detection of support for the NACK mechanism may be beneficial in some certain circumstances. For example, with the current definition of SigComp, acknowlegement of state receipt is required before a sender can reference such state. In cases in which multiple messages are sent before a response is received, the need to wait for such responses can cause significant decreases in message compression efficiency. If it is known that the receiver supports the NACK mechanism, the sender can instead optimistically assume that the state created by a sent message has been created, and is allowed to be referenced; if such an assumption turns out to be false (due to, for example, packet loss or packet reordering), the sender can recover upon receipt of a NACK.

In order to facilitate such detection, implementations that will send NACK messages upon decompression failure MUST set the least significant bit of memory position 11 to "1" when initializing their UDVMs. The bytecodes sent to such an endpoint can check whether this bit is set, and send appropriate indication back to their compressor as requested feedback. The other bits of bytes 10 and 11 are reserved for future extensions and MUST be ignored for the purpose of detection of NACK support.

> Open Issue: Is the above behavior the best way to detect support? Even without this additional behavior, it is trivial to probe for NACK support by sending a message intentionally designed to fail (e.g. message format 1 with random data for the state identifier), and check whether a NACK is received in response. The downside to such probing, of course, is that doing so adds another round-trip of messages when communication is initiated. Further, if a response to the probe is not received on an unreliable transport, the endpoint performing the probe has no clear way to determine whether the absence of a response is due to lack of support by the remote endpoint, or due to packet loss. Consequently, such a probe mechanism would require repeated retransmissions if no response is received.

3. Message Format

SigComp NACK packets are syntactically valid SigComp messages which have been specifically designed to be safely ignored by implementations that do not support the NACK mechanism.

In particular, NACK messages are formatted as the second variant of a SigComp message (typically used for code upload) with a "code_len" field of zero. The NACK information (message identifier, reason for failure, and error details) is encoded in the "remaining SigComp message" area, typically used for input data. Further, the "destination" field is used as a version identifier to indicate which version of NACK is being employed.

3.1 Message Fields

Although the format of NACK messages are the same as the second variant of normal SigComp messages, it is useful to demonstrate the specific fields as they appear inside the "returned feedback item" field.

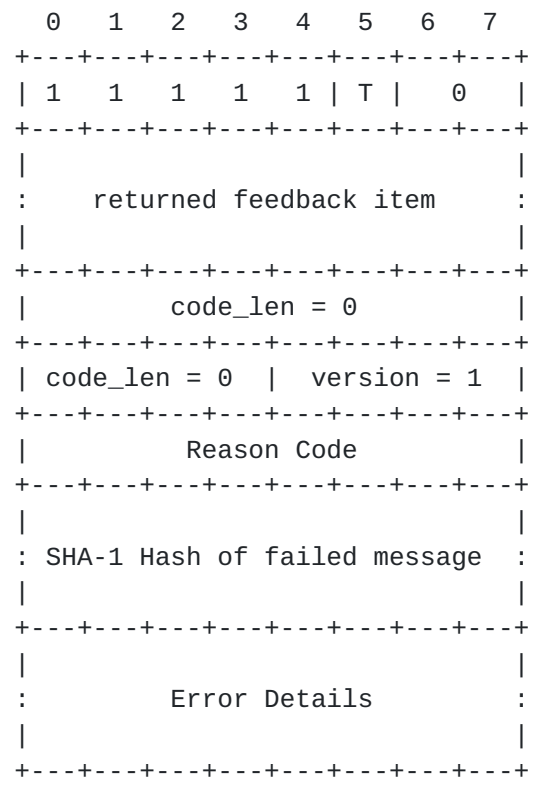

Figure 1: SigComp NACK Message Format

- "Reason Code" is a one-byte value that indicates the nature of the decompression failure. The specific codes are given in section Section 3.2

- "SHA-1 Hash of failed message" contains the full 20-byte SHA-1 hash of the SigComp message that could not be decompressed. This information allows the NACK recipient to locate the message that failed to decompress so that adjustments to the correct compartment can be performed. When performing this hash, the entire SigComp message is used, from the header byte (binary 11111xxx) to the end of the input. Any lower-level protocol headers (such as UDP or IP) and message delimiters (the 0xFFFF that marks message boundaries in stream protocols) are not included in the hash. When used over a stream based protocol, any 0xFFxx escape sequences are un-escaped before performing the hash operation.

- "Error Details" provides any additional information that might be

useful in correcting the problem that caused decompression failure. Its meaning is specific to the "Reason Code". See section Section 3.2 for specific information on what appears in this field.

- "Code_len" is the "code_len" field from a standard SigComp message. It is always set to "0" for NACK messages.

- "Version" gives the version of the NACK mechanism being employed. This document defines version 1.

3.2 Reason Codes

Note that many of the status codes are more useful in debugging interoperability problems than with on-the-fly correction of errors. The "STATE_NOT_FOUND" error is a notable exception: it will generally cause the NACK receipient to encode future messages so as to not use the indicated state.

Upon receiving the other status messages, an implementation would typically be expected to either use a different set of bytecodes or, if that is not an option, to send that specific message uncompressed.

Error	Code	Details
STATE_NOT_FOUND	1	State ID (6 - 20 bytes)
CYCLES_EXHAUSTED	2	Cycles Per Bit (1 byte)
USER_REQUESTED	3	
SEGFAULT	4	
TOO_MANY_STATE_REQUESTS	5	
INVALID_STATE_ID_LENGTH	6	
INVALID_STATE_PRIORITY	7	
OUTPUT_OVERFLOW	8	
STACK_UNDERFLOW	9	
BAD_BITORDER	10	
DIV_BY_ZERO	11	
SWITCH_VALUE_TOO_HIGH	12	
TOO_MANY_BITS_REQUESTED	13	
INVALID_OPERAND	14	
HUFFMAN_NO_MATCH	15	
MESSAGE_TOO_SHORT	16	
INVALID_CODE_LOCATION	17	
BYTECODES_TOO_LARGE	18	Memory size (2 bytes)
INVALID_OPCODE	19	
ID_TOO_SHORT	20	State ID (6 - 19 bytes)
ID_NOT_UNIQUE	21	State ID (6 - 20 bytes)
MULTILOAD_OVERWRITTEN	22	
STATE_TOO_SHORT	23	State ID (6 - 20 bytes)

Only the six errors "STATE_NOT_FOUND", "CYCLES_EXHAUSTED", "BYTECODES_TOO_LARGE", "ID_TOO_SHORT", "ID_NOT_UNIQUE", and "STATE_TOO_SHORT" contain details; for all other error codes, the "Error Details" field has zero length.

Figure 2: SigComp NACK Reason Codes

1. STATE_NOT_FOUND
   A state that was referenced (either using STATE-ACCESS instruction or in the actual SigComp message itself) cannot be found. The "details" field contains the state identifier for the state that could not be found.

2. CYCLES_EXHAUSTED
   Decompression of the message has taken more cycles than were allocated to it. The "details" field contains a one-byte value that communicates the number of cycles per bit. The cycles per

bit is represented as an unsigned 8-bit integer (i.e. not encoded).

3. USER_REQUESTED
   The DECOMPRESSON-FAILURE opcode has been executed.

4. SEGFAULT
   An attempt to read from or write to memory that is outside of the UDVM's memory space has been attempted.

5. TOO_MANY_STATE_REQUESTS
   More than four requests to store or delete state objects have been requested.

6. INVALID_STATE_ID_LENGTH
   A state id length less than 6 or greater than 20 has been specified.

7. INVALID_STATE_PRIORITY
   A state priority of 65535 has been specified when attempting to store a state.

8. OUTPUT_OVERFLOW
   The decompressed message is too large to be decoded by the receiving node.

9. STACK_UNDERFLOW
   An attempt to pop a value off the UDVM stack was made with a stack_fill value of 0.

10. BAD_BITORDER
    An INPUT-BITS or INPUT-HUFFMAN instruction was encountered with the "input_bit_order" register set to an invalid value (i.e. one of the upper five bits is set).

11. DIV_BY_ZERO
    A DIVIDE or REMAINDER opcode was encountered with a divisor of 0.

12. SWITCH_VALUE_TOO_HIGH
    The input to a SWITCH opcode exceeds the number of branches defined.

13. TOO_MANY_BITS_REQUESTED
    An INPUT instruction was encountered that attempted to input more than 16 bits.

14. INVALID_OPERAND

An operand for an instruction could not be resolved to an integer value (e.g. a literal or reference operand beginning with 11111111).

15. HUFFMAN_NO_MATCH
The input string does not match any of the bitcodes in the INPUT-HUFFMAN opcode.

16. MESSAGE_TOO_SHORT
When attempting to decode a SigComp message, the recipient determined that there were not enough bytes in the message for it to be valid.

17. INVALID_CODE_LOCATION
The "code location" field in the SigComp message was set to the invalid value of 0.

18. BYTECODES_TOO_LARGE
The bytecodes that a SigComp message attempted to upload exceed the amount of memory available in the receiving UDVM. The details field is a two-byte expression of the DECOMPRESSION_MEMORY_SIZE of the receiving UDVM. This value is communicated most-significant-byte first.

19. INVALID_OPCODE
The UDVM attempted to identify an undefined byte value as an instruction.

20. ID_NOT_UNIQUE
A partial state identifier that was used to access state matched more than one state item.

21. ID_TOO_SHORT
A unique state item was matched but fewer bytes of state ID were sent than required by the minimum_access_length.

22. MULTILOAD_OVERWRITTEN
A MULTILOAD instruction attempted to overwrite itself.

23. STATE_TOO_SHORT
A STATE-ACCESS instruction has attempted to copy more bytes from a state item than the state item actually contains.

4. Security Considerations

4.1 Reflector Attacks

Because SigComp NACK messages trigger responses, it is possible to trigger them by intentionally sending malformed messages to a SigComp implementation with a spoofed IP address. However, because such actions can only generate one message for each message sent, they don't serve as amplifier attacks. Futher, due to the reasonably small size of NACK packets, there cannot be a significant increase in the size of the packet generated.

It is worth noting that nearly all deployed protocols exhibit this same behavior.

4.2 NACK Spoofing

Although it is possible to forge NACK message as if they were generated by a different node, the damage that can be caused is minimal. Reporting a loss of state will typically result in nothing more than the re-transmission of that state in a subsequent message. Other failure codes would result in the next message being sent using an alternate compression mechanism, or possibly uncompressed.

Although all of the above consequences result in slightly larger messages, none of them have particularly catastrophic implications for security.

Normative References

[1] Price, R., Bormann, C., Christoffersson, J., Hannu, H., Liu, Z. and J. Rosenberg, "Signaling Compression", RFC 3320, January 2003.

[2] Hannu, H., Christoffersson, J., Forsgren, S., Leung, K., Liu, Z. and R. Price, "Signalling Compression (SigComp) - Extended Operations", RFC 3321, January 2003.

Non-Normative References

[3] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

Author's Address

Adam Roach
dynamicsoft
5100 Tennyson Pkwy
Suite 1200
Plano, TX 75024
US

EMail: adam@dynamicsoft.com

Appendix A. Comments and Feedback

Editorial comments should be directed to the author at adam@dynamicsoft.com. Discussion of the mechanism described in this document should be directed to the ROHC mailing list (rohc@ietf.org).

Appendix B. Changes

- Moved NACK parameters to end of message, so that NACK messages cen be distinguished from standalone feedback messages
- Changed behavior of endpoint receiving a NACK for a message sent on a reliable transport.
- Clarified some of the motivating text relating to server failover
- Added mechanism for detection of NACK support

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.