2009

A New covert channel over RTP

Christopher Forbes

A New Covert Channel over RTP

By

Christopher R. Forbes

Thesis submitted in partial fulfillment of the requirements
for the degree of
Master of Science in
Computer Security and Information Assurance

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

August 21, 2009

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Computer Security and Information Assurance

Thesis Approval Form

Student Name: Christopher R. Forbes

Thesis Title: A New Covert Channel over RTP

Thesis Committee

Name	Signature	Date
Bo Yuan Chair		8/28/09
Bill Stackpole Committee Member		8/28/09
Pete Lutz Committee Member		8/28/09

Thesis Reproduction Permission Form

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

Master of Science in
Computer Security and Information Assurance

A New Covert Channel over RTP

I, Christopher R Forbes, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction must not be for commercial use or profit.

Date: 8/28/09 Signature of Author: Christopher R Forbes

Abstract

In this thesis, we designed and implemented a new covert channel over the RTP protocol. The covert channel modifies the timestamp value in the RTP header to send its secret messages. The high frequency of RTP packets allows for a high bitrate covert channel, theoretically up to 350 bps. The broad use of RTP for multimedia applications, including VoIP, provides plentiful opportunities to use this channel. By using the RTP header, many of the challenges present for covert channels using the RTP payload are avoided.

Using the reference implementation of this covert channel, bitrates of up to 325 bps were observed. Speed decreases on less reliable networks, though message delivery was flawless with up to 1% RTP packet loss. The channel is very difficult to detect due to expected variations in the timestamp field and the flexible nature of RTP.

Table of Contents

1. Introduction

A covert channel provides discrete communications using data in motion as its carrier. A covert channel is hidden within a legitimate communications channel, and seeks transmission without detection of the channel's existence. This project will develop a covert channel using Real-time Transport Protocol (RTP) used during a Voice over IP (VoIP) call as the carrier. The high frequency of packets for multimedia communications using RTP (such as VoIP) makes it an attractive carrier for a covert channel. This covert channel will utilize the least significant bits of the timestamp in the protocol header to deliver its message rather than delivering it in the payload. By using the protocol header, the channel provides for broad applicability by ignoring many of the codec issues encountered in the payload. A sample implementation of the covert channel was also developed and experimental results taken.

With VoIP communications, the focus is often on the signaling protocols used, such as SIP and H.323. However, the signaling protocols actually form a small amount of the overall data traffic generated. Rather, the most common protocol by frequency is RTP, which is used to carry the audio data. RTP is used generically to carry multimedia for a number of streaming communications, including video and audio applications. This broad application gives an RTP based channel uses beyond VoIP, allowing for even wider usage. Prior research has focused on the payload, but the variable nature of the codecs used and possible re-encoding has been an issue. By moving the channel to the protocol header, the payload associated issues are eliminated at the cost of decreased bandwidth.

The first section reviews similar literature and details how they relate to our design. Next the design and methodology of the channel will be introduced. Following that, the reference implementation and a discussion of the results will be presented. Last, we will identify areas for future research and conclude.

2. Background

2.1 Prior Research

Given the rising popularity of VoIP, a number of papers have addressed covert channels within a VoIP carrier. Prior research has focused on embedding within the RTP payload rather than the header. The payload has many traits that make it attractive: a large amount of data to embed within, noisy data source, and the ability to change the payload without obvious detection. However, these existing channels have not been without problems.

One issue that was encountered was the lack of reliability, as RTP uses UDP as its transport protocol. To address this, protocols within the channel were used to increase reliability. Druid [1] addressed this by providing his own minimalistic protocol to be used within the covert channel. The author used simple protocol containing fields including checksum, type, length, and sequence fields. He makes use of control messages separate from data messages as opposed to sending control messages with the covert data. The design presented by Druid was effective, but could be costly in a low bandwidth channel.

The RTP payload based covert channel was not without its problems. One issue encountered by Druid [1] was the varying codecs that can be used within RTP, and can force a different solution for each codec encountered. This made implementations specific to a particular setup. Furthermore, codecs could be changed by media gateways, and the covert data could be lost by the re-encoding.

Another implementation of a covert channel in the RTP payload was presented by "An assessment of VoIP covert channel threats" [2]. The authors sought to use VoIP, in particular the RTP payload, to embed a secondary data stream hidden from obvious detection. They also developed a performance evaluation framework to be able to quantify the threats to users. They found that covert channels are well-supported within a VoIP carrier channel, and suggested that detection of these covert channels is possible. An implementation was also provided, showing the ability to embed a secondary voice stream in a RTP stream. However, they did little to address the major issues of reliability and integrity across an unreliable medium.

Other issues encountered with an RTP based covert channel include detectability and encryption. In “An M-Sequence Based Steganography Model for Voice over IP” [3] the authors seek to provide a real time steganography design that resists detection. They make use of the G.729 codec as a cover medium and established an m-sequence technique to hide the data. A RSA-like key exchange provides for synchronization between the two endpoints. The authors found that their techniques provided good security for transmitting covert data while maintaining the real time requirements of VoIP. The real time requirements are important to consider, as only a minor processing delay can be afforded between both ends of the channel without becoming easily detectable by ear. This was a novel approach but again focused on implementation in the RTP payload specifically using the G.711 codec.

Another implementation within a VoIP call was presented by “Covert channel for improving VoIP security” [4]. The authors sought to exchange information over VoIP using digital watermarking and steganography techniques to provide a covert channel, trying to provide a covert channel that could improve security of VoIP calls or provide bandwidth savings. To provide their solution, the authors embedded watermarks and steganographic data to provide for authentication and integrity of the VoIP stream. Control fields were embedded into the existing protocol headers, while the data were embedded within the voice stream. However, the simple control information provided was embedded within available unused protocol headers, which is likely to increase detection. The authors found that the covert data effectively improved the security of the VoIP communications by providing authentication and integrity, and noted that it could also be used for other applications. This differs from our new covert channel as it focuses on improving security for the call versus just developing a covert channel. They also have a less robust system for their data transmission. However, the idea of using protocol headers for some control information is shared, though the approach differs notably.

Other approaches have also been taken to use RTP as a covert channel. “Hiding Data in VoIP” [5] compares steganographic techniques that can be used to introduce covert channels within VoIP and provides a brief description of these methods and their characteristics. The primary method introduced makes use of both timing delays and modifying packet contents for a hybrid channel. The idea is to use excessively delayed packets that are discarded to carry a load of covert data. They found that this methodology provided relatively good cover against

detection, given normal delays of packets and resistance to statistical analysis. This differs notably from our new covert channel, due to the use of timing delays and using large portions of packet payload rather than sending a small amount of data in packets that are accepted by the receiver.

Another approach to a VoIP covert channel was using a full timing channel as presented in "Tracking Anonymous Peer-to-Peer VoIP Calls on the Internet" [6]. The authors investigated how watermarking can be used to track encrypted, peer to peer VoIP calls even in the presence of anonymizing networks. To accomplish this, they presented a way to embed a watermark in the VoIP stream which could be used for identifying the stream using a timing based channel. The authors found that it was possible to make timing adjustments that were small enough to allow for proper VoIP communications while allowing for their covert signal to be received. They also found that low-latency anonymizing networks are vulnerable to timing based covert channels. This solution only embedded a very small amount of data over the channel, but the persistence of timing across anonymizing networks is notable. This timing channel shows an alternative solution to using the RTP payload, but is limited in its data transmission capability.

A covert channel using the TCP timestamp was presented in "Covert Messaging Through TCP Timestamps" [7]. This channel carried 1bit in the least significant bit of the TCP timestamp by delaying the packet creation. It is essentially a timing channel, with the timing change read from the timestamp value. In TCP, the timestamp is only an option, making it easier to detect if a system does not usually use the option. In contrast, RTP makes use of the timestamp in every

packet. The TCP timestamp is set from the system clock, but has no set unit of measure (varies by operating system). This variability works favorably for the channel, as it makes detection more difficult. The data is obscured by an XOR against the 9th bit of the hash of the packet headers and the secret key, providing a simple yet reliable obfuscation method.

TCP provides several benefits over UDP, including the ordering of segments and reliability of its payload. However, the header of the TCP segment does not share this reliable delivery. In the case of a retransmission, the payload will be resent, but a new timestamp will be included. This means that the covert channel would need to watch for retransmissions of TCP and resend the lost data to take advantage of TCP's reliability. Instead, the authors send each bit multiple times to achieve reliable delivery. However, this further reduces the available bandwidth, leaving the channel with a very limited bandwidth that could restrict its usefulness for many scenarios.

2.2 RTP

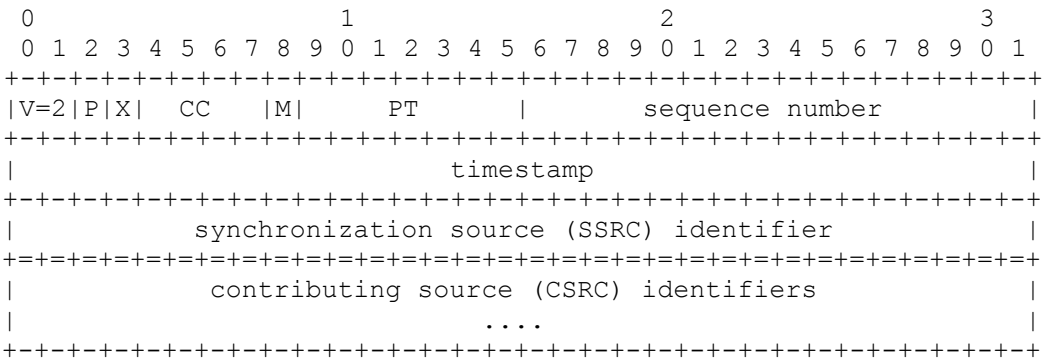


Figure 1: RTP Packet Header. Source: RFC 3550 [8].

The RTP packet header is shown in Figure 1 above. The smaller sized fields in the header, covering the first 16 bits of the header, are expected to have specific values for an RTP header. This means that making changes to these fields could be easily detected. Additionally, the small size of these first six fields greatly limits the amount of data that could be transmitted within them. The fields exceeding one byte in the RTP header are the sequence number, timestamp, SSRC, and CCRC. The SSRC and CCRC have set values for a given data stream, making any modifications to these values obvious. Also, the CCRC is often blank, and making any changes to an empty field is easily detectable. The sequence number must increment by a value of one for each RTP packet and is used in loss detection, meaning that it cannot be modified either. However, the timestamp value does not require an exact number, but rather accepts a range, allowing our covert channel to be embedded there.

It is important to understand the respective roles of the RTP timestamp and sequence number. The timestamp is used to determine order of the packets for playback, and with jitter calculations. If a high bitrate video is transmitted, it is possible for the timestamp to remain the same for several packets transmitting a single frame. In contrast, the sequence number is used to detect RTP packet loss, as they only increase by one for each packet transmitted. [9]

The timestamp in the RTP header is not derived from actual time values in most cases, but rather from the sample rate that is used by the codec in the RTP payload. The most common sampling rate, 8 kHz, results in a timestamp increment of 160 for most codecs. Similarly, a wideband 16 kHz sampling results in a timestamp increment of 320 per packet. This is based on

the number of samples per frame, and with most codecs covering 20 ms/packet. 20/1000 ms * 8000 Hz = 160 samples per frame. Accordingly, the timestamp unit for 16 kHz wideband codecs is 16,000/50, which equals 320. Units are not usually used in reference to the RTP timestamp, given its somewhat arbitrary nature, but are equivalent to samples/frame. Note that there is a special exception with G.722, which is actually 16,000 Hz but registered as 8,000 Hz by mistake in an RFC, so it actually uses a 160 increment instead of 320. [10]

Accordingly, codecs with alternative sampling rates can affect the frequency at which RTP packets are sent, and their corresponding timestamp interval. This means that the range available for covert channel use varies with the codec used. Based upon common voice sampling rates, most codecs use a timestamp increment of 160 or 320, either of which would work using this new design. If a codec were encountered which did not use a multiple of 160 in the timestamp, then modifications would have to be made to accommodate this. By designing the channel to expect only a timestamp increment of 160, it increases its flexibility for being used with a wide variety of codecs at common voice sampling rates.

Name	Standardized by	Description	Bit rate (kb/s)	Sampling rate (kHz)	Frame size (ms)
(ADPCM) DVI	Intel, IMA	ADPCM	32	8	sample
G.711	ITU-T	Pulse code modulation (PCM)	64	8	sample
G.721	ITU-T	Adaptive differential pulse code modulation (ADPCM)	32	8	sample
G.722	ITU-T	7 kHz audio-coding within 64 kbit/s	64	16	sample
G.722.1	ITU-T	Coding at 24 and 32 kbit/s for hands-free operation in systems with low frame loss	24/32	16	20
G.723	ITU-T	Extension of G.721 adaptive differential pulse code modulation to 24 and 40 kbit/s for digital circuit multiplication equipment application	24/40	8	sample
G.723.1	ITU-T	Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s	5.6/6.3	8	30
G.726	ITU-T	40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)	16/24/32/40	8	sample
G.727	ITU-T	5-, 4-, 3- and 2-bit/sample embedded adaptive differential pulse code modulation (ADPCM)	var.	?	sample
G.728	ITU-T	Coding of speech at 16 kbit/s using low-delay code excited linear prediction	16	8	
G.729	ITU-T	Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear-prediction (CS-ACELP)	8	8	10
GSM 06.10	ETSI	RegularPulse Excitation LongTerm Predictor (RPE-LTP)	13	8	22.5
LPC10e (FIPS 1015)	US Govt.	Linear-predictive codec	2.4	8	22.5
Speex (narrowband)	Xiph.org	Open source codec based on CELP	2.15-24.6	8	20
Speex (Wideband)	Xiph.org	Open source codec based on CELP	4-44.2	16,32	20

A table of common codecs used in VoIP, with sampling rates shown.
Sources: Henning Schulzrinne [11], speex.org [12] [13], IANA[14].

3. A New Covert Channel over RTP

3.1 Covert communications in RTP timestamp

This project will modify the timestamp field in the RTP header and use it to transmit covert data. For a regular voice stream sampled at 8000 Hz using G.711, the timestamp is incremented by 160 in each packet rather than incrementing by the actual time passed. The 160 is based on the expected time between packets based on the sample rate. However, the numbering of the packets needs only to be in proper sequence to function properly at the receiving end.

The G.711 codec was chosen as the reference point due to its standardized nature, high quality performance, and common usage in VoIP communications. G.711 provides a high quality communications channel, using pulse code modulation at 8,000 Hz with 8 bits per sample for a 64 Kbit/s bit rate [15]. G.711 μ-law was used, as this is the version commonly used in North America. G.711 is typically used for VoIP setups when bandwidth limitations are not an issue due to its high quality sound.

Data can easily be embedded into the last seven bits based on standard VoIP sampling rate of 8 kHz without disturbing transmission. Seven bits cover a range of 128 (0-127) in decimal, which is still below the value of 160 used for incrementing the timestamp. With 50 packets transmitted per second, this provides a gross data rate of 350 bits per second full duplex, with less available dependent on network conditions, reliability protocols implemented, and needed level of covertness.

While this throughput is less than the 1000 bytes established in [1] using the RTP payload, this is to be expected given the limited size of the timestamp (4 bytes) as compared to the payload size with G.711 (160 bytes). In addition, this new method makes use of a higher percentage of bits available in its respective field, using 21.9% of the field as opposed to the 12.5% used in the payload of [1].

3.2 Challenges

There are numerous challenges to making this covert channel successful. The first is that the transport layer for RTP, UDP, is connectionless and unreliable. It performs best effort delivery, but there is no retransmission of corrupt or missing data. Data may arrive out of order, corrupt, or not arrive at all. Accordingly, to ensure reliable transit, our covert channel must provide its own mechanism.

The next challenge is limitations imposed by cover size. In our particular case, the RTP packet is composed of 32 bits of data. Of those 32 bits, 7 can be safely used for the covert channel. This small size per packet of this channel means that messages must be spread across multiple packets and reassembled properly at the other end. The dependency on a high number of packets means that the threat of packet loss is multiplied, especially for delivering data intolerant of any loss. It also means that efficiency is of the utmost importance, as there is little room for any overhead in the covert channel.

Latency can also prove to be a challenge, given the real time nature of the application. Sensitivity to network latency is very noticeable on a phone call, as too much latency is easily noticed by end users and can render a call unusable. This provides a challenge in that the

additional overhead from embedding and receiving message cannot delay RTP by much, or else the covertness is lost.

The nature of the RTP streams can also prove to be problematic. RTP uses two separate half-duplex streams, rather than a single, full-duplex stream. The split nature makes tracking stream progress more difficult. This means that both RTP streams must be correlated and used in support of the full-duplex solution for the covert channel.

In addition to common RTP issues, there are also specific RTP Payload issues. Payload based solutions suffer from compression issues, particularly with the lossy compression that is commonly used. In addition, different audio codecs may require different approaches to work properly. Also, media gateways may re-encode audio or otherwise modify the RTP payload, thereby resulting in data loss. By using the timestamp, our covert channel largely bypasses codec issues. However, since timestamp is derived from the codec sampling rate, any frequency that is not a multiple of 8,000 Hz could require some reworking.

3.3 Characteristics of the Covert Channel

This covert channel makes use of the timestamp of RTP for carrying the data, rather than being used in support of another carrier as was done in other covert channels. In our implementation, the timestamp will pass accuracy checks, instead of failing checks to deliver a covert payload. In addition, there is no dependency upon the rest of the header or the payload for the covert channel to function. This simple approach makes it easier to implement reliably, with fewer bits of data that could potentially be in error.

3.3.1 Covertness

This implementation is difficult to detect provided the channel is not already known. The timestamp field is expected by most to contain actual timestamp values, which do not follow an exact pattern but rather contain some variability. However, if it is known that the RTP timestamp used for a particular stream increments by a certain amount, detection could focus on looking for variations from this expected increment. Such detection attempts could suffer from false positives, given the unreliable nature of UDP and the potential for datagram loss or disorder. Furthermore, the particular usage of RTP needs to be identified in order to know the expected increment, including not only the application but the particular codec in use for that particular stream. It is also difficult to see the data when examining packets, as the values are added to the timestamp as opposed to replacing the least significant bits. Detection of covert channels based on RTP focuses on the payload manipulation, which is left untouched for this channel.

The varying timestamps by codec also is advantageous as it makes it very difficult to detect manipulation of the timestamp. Since even a single application (VoIP) makes use of a wide range of codecs, simply detecting that timestamps are not incrementing at a certain rate is not easily accomplished without very heavy and specific processing, based on every possible known codec. This requires identification in the data stream of the codec used, which can be accomplished via the codes as detailed in RFC 3551 (G.711 ulaw is 0) [10]. However, knowledge of RTP usage and of the codec is required for this kind of detection to work. Additionally,

newer codecs, such as speex, do not even have an assigned number for RTP payloads, further complicating detection.

Another difficulty in detecting the RTP timestamp is that UDP is not reliable, meaning packets may be lost in transit. This results in packets that appear to not follow a regular pattern. In addition, it is possible for RTP streams to be reset during a call, resulting in the timestamp to be reset to a new start time. Both of these aspects of the RTP stream add to the irregularity of the channel and impair detection.

It can be difficult to even detect that RTP is in use. UDP does not specify what kind of data is in its payload. To be certain that RTP is in use, the start of the call must be captured, where the signaling protocol specifies a RTP transport. Wireshark [16] is able to detect an RTP stream if it catches the call setup, but otherwise will display as only UDP. When using the SIP protocol, the use of RTP is specified in the SDP in the message body of the SIP 200 (ok) packet (see Appendix 1). If capturing packets after the call has started, they will also only show as UDP in Wireshark. These issues mean that filtering for the channel faces many difficulties, especially since RTP need not use ports around 4000 as is customary. Furthermore, if a packet capture is saved without the session initiation, the RTP packets will again only display as UDP. The RTP header does not make it obvious it is in use, as the only static field is the first two bits, representing the RTP version. With only two bits in use, there is a 1 in 4 chance that any packet will match that pattern, making it a very poor method. Since it can be difficult to even detect RTP usage, looking inside of it for a covert channel becomes even more difficult.

There is also a matter of where the endpoints of the covert channel are in comparison to the RTP stream. To detect the covert channel, one must be able to capture packets from a point where the covert channel is present. By using a man in the middle approach, the covert data can be added after it leaves one client and stripped before it arrives to the other client. This means that at the end points, there is no evidence that the covert channel ever existed. Enterprise perimeters can only detect it if one of the covert channel interceptors is within their perimeter; otherwise, they are completely useless as well. Granted this requires the covert operators to gain network access en route, but this is not an impossible task.

The ease of detecting the channel also depends upon how much data is being transferred. The more data that is transferred, the more irregular the timestamp may appear. While it is unlikely that the timestamp is examined in detail for the call, it is possible to do so. Thus performing a large file transfer will create many modified packets, while a few lines of text sent back and forth modifies very few of the RTP packets. More covert data packets also mean greater entropy in the timestamp increment, which could aid in detection.

3.3.2 Reliability

The channel is not particularly robust without further mechanisms being used to make it a reliable carrier. This is because it is carried over a UDP carrier, as opposed to a reliable carrier such as TCP. As such, part of the available bandwidth can be dedicated to a minimalistic protocol to improve the reliability and integrity of the messages. However, given the limited bandwidth available, the channel may still experience reliability issues when faced with a highly disruptive channel. In the reference implementation, transmissions functioned without error in

the majority of the tests, even with imperfect network transmissions. The protocols used for increasing reliability are detailed within the experimental design section.

3.3.3 Bandwidth

The bandwidth of the channel is up to 7 bits per packet by 50 packets a second, for a total of 350 bits per second. This max rate can be closely realized using the raw data transmission on a well-functioning local network. When using the reliability protocol in the reference implementation, this rate was in the range of 60-130 bps. This drop in speed is largely dependent upon distance, as increased distance lengthens travel time for the packets and their acknowledgments. The speed from the channel is still sufficient for text based communications and small file transfer, among other applications. However, the speed from the reliability mode represents a significant drop from the raw data transmission rate.

The main downfall of the reliability protocol used is that its acknowledgment system allows for only one piece of data to be sent before an ACK is received. This greatly slows down the data transmission, as it needs to wait for the data to get to the receiver and for an ACK to travel back before any data transmission can occur. During this round trip travel time, no additional data may be transmitted. It is important to note that this is true for each direction independent of the other; each direction ACK and transmit is almost entirely independent of the other direction. The only connection is that the ACK for one direction is sent before new data going the opposite direction. While this seemingly could allow one sender to absorb all of the data transmission, in reality that is not the case. The fact that only one piece of data is sent at a time in each direction provides for abundant unused packets, simply based on the time the data is

on the wire. While this underutilizes bandwidth, it improves ordering and reliability of the stream.

An alternative considered was to use error correction code to detect packet loss and perform data reconstruction rather than using an acknowledgment based system. This would allow for higher speeds, with only a small amount of data being used for reliability, and put no holds on the data transmission rate. This also runs the slight risk of packets arriving out of order, which is detrimental to data reconstruction. While packets could be ordered based on the RTP sequence number, this further complicates this setup. Also, this solution places no guarantees against data loss, though it is effective against periodic loss of a single packet. However, multiple packets missing in short sequence would result in lost data with no hope of recovery. Given this shortfall, the acknowledging system was chosen instead.

4. Experimental Design

4.1 Methodology

In order to transmit the data over the RTP stream, it first needs to be known the rate at which packets are sent. This design focuses on RTP as used for VoIP application, but it could be used for other RTP streams as well. With VoIP at 8000 Hz, the RTP timestamp increments by 160 in each packet, with packets sent 50 times per second. This means that up to 7 bits in the timestamp can be utilized to carry data for the channel (128 decimal). With roughly 50 packets per second, this provides a gross data rate of 350 bits per second. It is unlikely that this full speed would be realized, as it is a theoretical maximum, expecting packets to be sent at an exact speed of 50 packets per second with no data loss from end to end. In reality, data loss, protocol overhead, and other miscellaneous factors tend to slow the data rate.

As an alternative to the 7 bit transfer rate, a lower bit rate and / or introducing delays in transmitting data can increase the covert nature of the channel. Since the lower 5 bits do not change with all codecs, any changes in those may be more conspicuous. To counter this, use the 6^{th} and 7^{th} least significant bits avoids any simple detection of the channel while the data is in binary form. These two bits correspond to the values 32 and 64 in decimal. It also decreases the data rate and provides more order, which can decrease the entropy. However, this greatly reduces the available bandwidth to the channel, and requires that data be spread out across more packets. This method is best suited for situations were covertness is the highest priority.

To embed data for the covert channel, we take the timestamp value and add the value of the data to be transmitted, while subtracting any previously added value. For example, transmitting the value "A" is prior_ts-last_mod+160+65, with prior_ts being the last timestamp, and last_mod being the last timestamp modification used. This means that all necessary tracking of prior values is done by the sender, not the receiver, simplifying the process and avoiding the two RTP streams issues. It also means that, even with no additional reliability, the channel can communicate even in the result of packet loss. In order to keep the timestamp close to the expected value, removal of the previously transmitted value must be removed from the timestamp. This allows for the timestamp to stay closer to the actual value, thereby helping to avoid detection and potential software malfunction.

To decode the data, modular arithmetic is used. It is performed as x mod(160), where x is the timestamp received. This simplifies decoding and allows for reading the covert data even when experiencing packet loss. This method also supports random start points, provided the implementation uses the proper timestamp increment when choosing the start point. If a particular implementation does not do so, it is necessary to run the modular arithmetic above after subtracting the initial timestamp value.

There are numerous models under which this covert channel could be implemented. It can be added to a VoIP client, run as a standalone application on one of the computers a call is placed from, or run as a proxy/man in the middle. In addition to these modes, any combination thereof would work, as they all communicate to each other in the same fashion. What differs is

where the covert data is added and removed from the RTP stream, and the access required to use them.

The reference implementation was integrated into a softphone package that can serve as the basis for many other applications. This allowed for easy testing and demonstration of the project, as well as insight into the operation of the phone. By placing the channel in a common softphone, detection of the software would prove more difficult than an additional process on a local computer, and would not require the same kind of privileged access to use. However, the ultimate for covertness is man in the middle, provided proper network access can be obtained (and that is the difficult part). Use as a softphone also provides for the least latency, as there is no additional packet capturing and formation over that of the phone call. This is under the assumption that all implementations are written in languages and run on platforms with similar execution times. Also, UDP checksums need to be recomputed for a man in the middle implementation, adding further overhead.

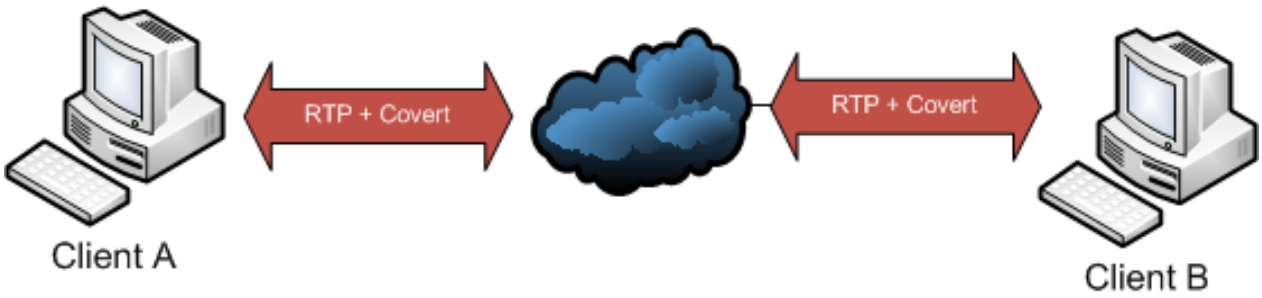


Figure 2: The covert channel being run on the endpoint computers, either as a standalone application or built into a softphone.

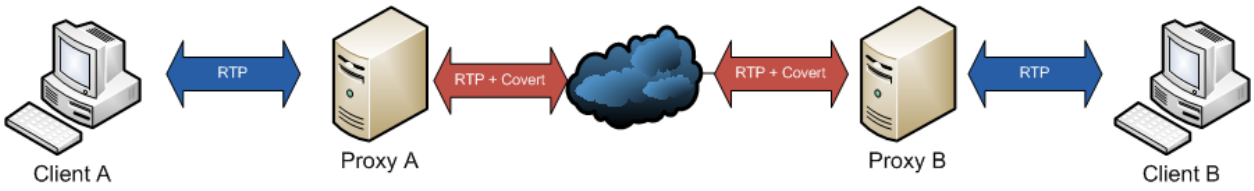


Figure 3: The covert channel being used via proxy servers.

The reference implementation was tested against several common codecs. All of the codecs used worked without issue with the covert channel. Most of the testing was performed against G.711, which uses a timestamp increment of 160 and is considered the baseline for quality voice audio. Other codecs used include G.722, which is actually 16,000 Hz but registered as 8,000 Hz by RFCs, and actually uses 160 as its increment. Speex, an open source solution, was used at 16,000 Hz, giving it a timestamp increment of 320. GSM, running at 8 kHz and a timestamp increment of 160, also performed admirably. The only major codec not tested with the implementation was G.729, which operates at 8 kHz and is heavily compressed. G. 729 is patent encumbered and considered costly, and accordingly was not included in the open source softphone that the reference implementation was built upon.

4.2 Minimalistic protocol for communications

In support of enabling communications using the RTP covert channel, a minimalistic protocol can be utilized to improve robustness of the channel. While UDP provides a checksum to verify the integrity of the data it carries, it does not guarantee delivery of data. Similarly, RTP provides sequence numbering for ordering packets, but does not guarantee reliable delivery either. Accordingly, what is needed for the covert channel is its own reliable transport protocol for use when packet loss is an issue or assured delivery is required.

In the case of a softphone implementation of the channel, the phone software is relied upon for verifying the UDP checksum. In other implementations, this would need to be performed by the covert channel implementation. In addition, depending on the protocol used, the implementation may need to check the RTP sequence numbering or timestamp to ensure proper ordering of the packets.

4.2.1 Simple Acknowledgment

In this implementation, a very simple acknowledgment (ACK) and retransmission system was used to improve reliability. Flowcharts of transmission and receiving with this protocol are shown in Appendices 2 and 3. When a packet is transmitted, a flag is set by the transmitter to not transmit again until an ACK is received. A two second timer begins to count as well, and if this is exceeded without receiving an ACK, the last piece of covert data is retransmitted. On the receiving end, when data is received (anything that is not an ACK or mod 160), a flag is set that an ACK needs to be transmitted in the next packet sent. Once this ACK is sent, the receiver can resume sending its next piece of data, provided it has received an ACK for its last piece of data.

The two second retransmission delay was meant to ensure adequate time for acknowledgments were provided. The buffer delay is only approximately 120 ms, but this came into effect on receipt of the data and receipt of the ACK, plus the time on the wire in both directions. It is likely that one second would suffice in most circumstances, but this could risk premature timeouts and make data collection difficult. This increased timer length can cause longer delays on noisy connections, but has no impact on speed unless retransmissions are

needed. For the reference implementation, it was preferable to eliminate premature timeouts as an issue.

In order to mitigate retransmissions due to lost ACKs, two ACKs are now sent in short sequence, with two different code values (see Appendix 4). This is meant to prevent retransmission of properly received data, resulting in duplicate data when an ACK is dropped. Since it is considered more likely for two subsequent packets to be lost than two non-sequential, the second ACK is transmitted in the second RTP packet after the first ACK. This also helps by making the channel more covert, as two sequential timestamp adjustments are easier to detect than two non-sequential packets. To avoid mistaking the second ACK as an ACK for new data, it is ignored if first ACK was received. The second ACK can be clearly identified since it has the sequence number of the first ACK + 2. If a second ACK is received that is not the prior ACK +2 in sequence, then it is used for the acknowledgment.

Integrity is provided for by the UDP checksum, which covers all of the data in its payload as well (including the RTP header). If the UDP checksum is valid, the datagram is kept and the RTP timestamp processed. If the UDP checksum is invalid, it is simply discarded, and a retransmission will occur if data was present in the header. No signaling goes back on a discard, since there is no way to determine if covert data was actually carried within that RTP packet or not.

Another available option is to send a packet before a retransmission, signaling that the next packet is retransmitted data. This could be effective in helping avoid duplicate data being

received, but it runs the risk of further complications with the retransmit packet itself being lost. In the reference implementation, this part of the protocol was not implemented.

Originally, the idea was to create a reliable system that was virtually flawless. However, given the small amount of space per packet, and chance of packet loss, a perfect solution was not devised for packet loss exceeding 1%. With more available space, one could reply back with the sequence number of a corrupted or lost packet, or note the sequence number a retransmission was sent to replace. However there is not that much space, even without transmitting any additional data, and spanning multiple packets would prove too unreliable for a reliability protocol.

4.2.2 Doubling

An alternative method considered is to limit the character set to base 64, and allow doubling to represent 64 bit retransmissions of the corresponding data. By doing this, it is easy to determine if the data sent should be accepted or not, based on whether an acknowledgment was sent or not and comparison to the last received data. If the retransmission is different than the last received, it should be kept and acknowledged. If it is the same, it may or may not be a duplicate, and still must be acknowledged. The default behavior is to presume it is a duplicate and discard it, though there is a chance that the original data was lost, and not the ACK. The likelihood of the ACK or data packet being lost is the same, but there is only theoretically a 1 in 64 chance that the second piece of data matches the first. Overall, there is a 1 in 128 chance of a duplicate occurring from this when a packet is lost. As such, this solution attempts to address the issue, but will not be entirely reliable in all cases.

On codecs using a longer timestamp interval (such as 320 when sampled at 16 kHz), another modification can be used to improve reliability while maintaining a larger amount of data to transmit. A similar doubling technique is used, but doubling the 128 basic ASCII values instead of base 64. This provides an estimated 1 in 256 chance of duplication should a packet be lost in one direction, which is better than the base 64 solution as well. However, this limits implementation to 16 kHz samplings when most voice is still sampled at 8 kHz, restricting its usefulness. The wider variations in timestamp increment also increases entropy, thereby limiting covertness.

4.2.3 Binary transfer

Another method to address potential reliability issues focuses on the actual binary transmissions. Instead of manipulating the timestamp value in decimal form, bits are manipulated directly. Two control bits are used as the protocol header, with 5 bits available for data transmission per packet. With roughly 50 packets per second, this provides a net data rate of approximately 250 bits per second and gross data rate of 350 bits per second. Since the RTP protocol already provides a sequence number, separate sequencing is unnecessary. The two bits are used to flag four different conditions: new data, stop, acknowledgment (ACK), and retransmit, as detailed below.

- New data: signals new data sent in the payload.
- Stop: signals the end of a data block, or no new data to send. This is sent in the packet after covert data was last transmitted in the payload. Arrival means no data

is available for retrieval in the payload. Regular timestamp data will be transmitted in the payload instead.

- ACK: sent in reply to acknowledge last transmission. Data is copied back to receiver for accuracy checking. If ACK is not received by the sender after a set time, data is sent again.
- Retransmit: set if ACK is not received within timer limit for acknowledging data. Payload contains last data transmitted.

5. Reference implementation

5.1 Implementation and environment

To implement this design, existing open source programs were modified to write packets to communicate the covert data. This was favored over starting from scratch, as it is redundant to start from scratch given the freely available software. Integrating the channel into the phone software eliminates an extra process and executable file that may raise detectability on the end user system. Additionally, this could minimize overhead for packet processing, since there is not the extra overhead of capturing and reforming packets. Depending where the covert channel is implemented, it can even take advantage of the jitter buffer for ordering of packets. However, a standalone application would provide a more universal application, free of constraints of a particular softphone, and could even implement its own jitter buffer. The current reference implementation was done in an existing softphone, though future versions will be done in standalone form as well.

C was chosen as the programming language due to its portability and speed. Java was looked at due to its ease of networking, but C offers faster execution and more low level access. In addition, stucts in C provide for an efficient and logical way for storing packet data. The major downside to C is that any GUI development is typically more heavily linked with particular operating systems than is Java.

5.1.1 Build Environment

The reference implementation was designed to run on Windows XP and Windows Vista. Other versions of Windows may work but have not been tested. The software was built in Visual Studio 2005, using the Visual C++ component. The Windows Platform SDK is necessary for the build environment, and the Windows SDK for Windows Server 2008 and .NET 3.5 (SDK 6.1) was used for the final versions of the implementation [17]. This allows for greatest compatibility with current Windows versions, including Windows XP, Windows Server 2003, Windows Vista, and Windows Server 2008. The DirectX SDK March 2009 was used for dependencies of its sound libraries [18]. The application may also build using Visual Studio 2008, though this is not a supported environment for PJSIP. Using Visual Studio 2005 and the newest version of PJSIP, the resulting program has been entirely stable, with no crashes occurring.

In addition to the Windows version, another version (using the same code) was built for Linux. This additional build was simple to do, as PJSIP includes make files for use with GCC on top of MS VC++ project files. The Linux version was compiled on Fedora 11, running Linux kernel 2.6.29.6, GCC 4.4.0, and GLIBC 2.10.1. These versions use an identical codebase, besides the parts that directly interface with OS API calls. Both builds of the reference implementation worked in the same fashion, though testing focused on the Windows build. The Linux build was primarily for showing that portability of the code works.

5.1.2 Base Project

The reference implementation of the project was being built upon the sample implementation provided by pjsip.org [19]. The application, pjsua, provides a basic SIP client based upon libraries from the PJSIP open source project. This provides an easy to use yet flexible VoIP application using SIP and RTP. Part of the appeal of this client is that it does not require the use of a PBX but rather can perform direct, peer-to-peer calls, thereby simplifying testing. However, it is also capable of placing and receiving calls by registering with a VoIP PBX. In testing, an Asterisk PBX was used as the soft PBX [20].

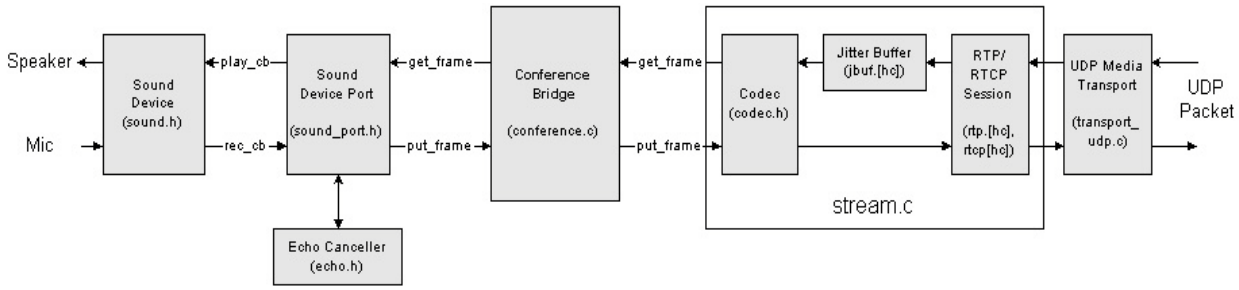


Figure 4: Media Flow of PJSIP [21].

The media flow of the PJSUA client is shown above. While all of the segments shown above are used in placing calls, the only parts involved in the covert channel are rtp.c and rtp.h (see appendices 7-9 for sources code). Accordingly, the vast majority of the application is not even aware of the timestamp modification. To further conceal the presence of the covert channel, the covert channel data is removed from the timestamp by rtp.c before heading to the rest of the application. By doing so, the modifications made by the covert channel are seen by as little of the application as possible, and do not interfere with the jitter buffer or other components.

While leaving the covert data did not appear problematic to the rest of the application, taking extra precautions will, if anything, result in better quality.

While removing the timestamp modification from the receiving end is not necessary for an implementation built into a phone client, it is more important when performing a man in the middle style operation. When performing the channel using a standalone application or a proxy, resetting the timestamp to the expected value can prevent the end user from ever having a chance to detect the channel. At first glance it seems that reverting timestamps could allow for a larger data amount sent per packet without disruption, but this requires the receiving end to track the last timestamp received (since modular 160 alone might give inaccurate results). While this could be accomplished, the increased complexity and potential for packet loss puts the reliability of the channel at risk. In addition, the covert channel would become easier to detect mathematically, given the larger fluctuations in timestamp increments.

The reference implementation currently provides a framework for transmitting data in ASCII form. Two main versions of the application were produced: one sending data without additional reliability mechanisms, and one following a basic protocol to improve reliability. The version without the reliability protocol enabled relies on very low packet loss to ensure delivery, or that a message can be useful without receiving every last character (text based communications). This “raw” version might work fine if the network connection is good along its entire length. For data where no loss is acceptable, or on noisy networks, the reliability enhanced version should be used.

The reference implementation also relies on reading values from one text file, and writing received values to another text file. This was done because the user interface was already crowded on the command line application, and it was not realistic to provide both reading and writing on the command line display. This is an area where a GUI could lead to improvement. However, to demonstrate that the data is being transmitted, received data is printed to the console on top of being written to the file (see Appendix 6). By default, these are named input.txt and output.txt, and should contain ASCII characters.

Line encoding (UNIX or Windows) does not make a difference, as newline and carriage return are simply transmitted as is any other character, printable or not. Since the characters are actually representative of an integer in C, the values can be read in as characters and added to the timestamp easily. Similarly, writing the value out is simply a matter of determining the timestamp modification and writing that as a character to the file. This keeps the process simple and efficient.

There are two values that are swapped for reasons of keeping the channel covert while on the line, the Null character (0) and no data (128). Since the desired effect is to have no timestamp modification when no covert data is sent, the value of 0 and 128 were swapped on the wire. These swaps are transparent to the read and write operations, as swapping occurs after reading and before writing. The decision was made to just move null rather than moving every character up by one for a range of (1-128) instead of (0-127), as this was considered more

efficient in processing. Besides "no data", the other special values transmitted are 130 and 131, which represent ACK 1 and ACK 2, respectively.

To transfer binary files, they were first encoded in base 64 to ensure reliable transfer. This is similar to the MIME encoding used in email, and is possible using a wide variety of tools. As this results in sending a 6 bit character as a 7 bit character, $1/7^{th}$ of the bits are wasted in this mode of transfer. Binary data could be encoded as ASCII prior to transfer without the overhead of base 64, but tools are not as readily available to perform this task.

The phone calls were placed using the modified version of PJSUA using the following configuration options for standard tests (see Appendix 6 for PJSUA main menu). Note that the play-file and auto-play flags were not always set, depending upon the particular test conducted.

	Command
Calling Phone	covert.exe sip:#.#.#.# --add-codec pcmu --no-vad
Receiving Phone	covert.exe --play-file noise.wav --auto-answer 200 --auto-play --no-vad

Covert.exe is the name of the executable file for the modified softphone. Sip:#.#.#.# contains the IP address of the answering phone in place of the pound signs. –add-codec pcmu specifies to place the call using G.711 U-law. –no-vad ensures the phone sends RTP packets, even when silence is detected. –auto-answer 200 has the phone receive calls automatically

and respond with SIP 200 (ok). –play-file noise.wav has the phone play back a sound file, and –auto-play has that sound file play back automatically when the call is established.

Most of the experimental results were collected running two instances of the client, one running on Windows Vista SP2 and the second on a Windows XP SP3 virtual machine. For thoroughness, tests for functionality were also run with a number of different XP and Vista machines at different locations, as well as Linux builds. Data was captured using Wireshark for analysis. In some instances, a sample noise file was used for the RTP data at one end, while in others microphones were utilized for both ends. The X-Lite 3.0 softphone [22] was also used as a secondary receiver for testing purposes, and a Grandstream Budgetone 200 SIP phone was used as a simple hardware based client for testing call functionality. However, all data rates and statistics were run from one covert client to another; other phones were used simply for interoperability testing. Unless otherwise noted, the G.711 codec was used for the RTP payload with a sampling rate of 8000 Hz.

For NY to IL tests, they were run between a Windows Vista SP2 machine and a Windows XP SP3 machine. Both machines were behind home routers using NAT, and connected to the Internet using cable modems. Both computers were connected to the LAN via 802.11G wireless networking. This wireless networking provided another source of potential interference for these tests, as multiple computers were using the shared wireless medium at both ends. Sample statistics from pjsua for a NY to IL call is shown in Appendix 5.

For the tests involving packet drops, they were run from a laptop running Windows Vista 64 bit to Windows Server 2008 64 bit. The two machines were connected to a switch using gigabit Ethernet. Percentage of RTP packets dropped was set on the application using the --tx-drop-pct=% --rx-drop-pct=%, where % is the percentage of packets to drop in that direction. Percentage drops were performed at 1, 3, 5, and 10 percent, with the percentage being the same for both receiving and transmitting.

Additional tests were performed XP to XP, Vista to Vista, and Linux to Vista, using various computers. These tests were not performed for speed but rather to test functionality of the product across a number of different systems. In all cases, the client was found to work identically regardless of the operating system used. Speed and reliability data was not collected from tests using Linux.

For the tests, the standard file used was a 1024 character block of ASCII text using Windows line endings. This file was transmitted in full duplex. While it is encoded in the file as 8 bits per character in ANSI for 256 possible characters, the ASCII character set records each character as only 7 bits for 128 possible values. Accordingly, the calculations for speed are based on the number of character in the file multiplied by 7, rather than 8 as is stored on the disk. This gives a more accurate representation of the data transmitted on the wire. In addition to the standard 1024 character file, a 300 KB file was also transmitted on some local tests to check for the ability to send large files.

5.1.3 Alternative implementations

In addition to the two primary versions of the reference implementation, two other versions were created. The first is the original acknowledging system, which makes use of a single acknowledgment. The one ACK mode, like the two ACK build, is designed to optimize reliability of the channel. The second alternate mode increases covertness by modifying only the 6th and 7th least significant bits of the timestamp (the 2 bit mode). While this reduces the overall bandwidth, it avoids detection by not changing bits that are usually unmodified in the timestamp. It also reduces the entropy of the overall channel. Recall that the second main implementation is a raw transfer mode, which sends data without any acknowledging or reliability measures to maximize bandwidth. This means there are the ACK based modes favoring reliability, the 2 bit mode favoring covertness, and the raw mode favoring bandwidth.

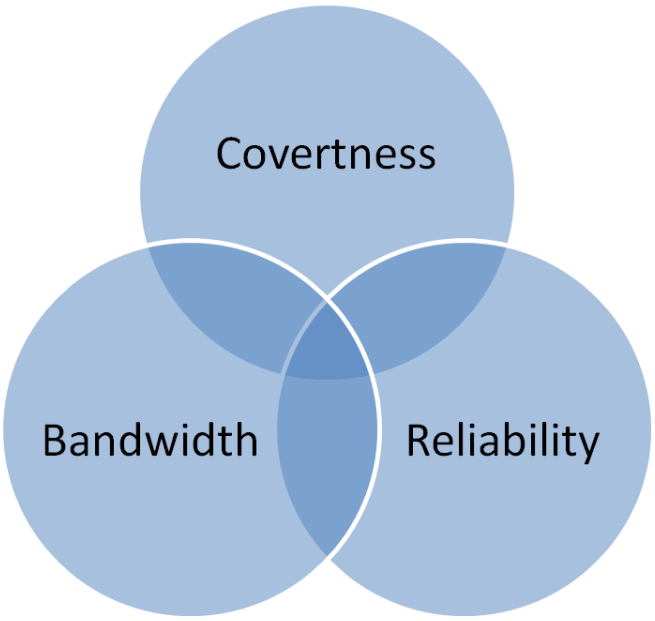


Figure 5: Components to balance in a covert channel

Depending upon the situation, one of these may prove to be better than another. For legitimate uses of a secondary stream, the acknowledging or raw modes should be used. For maximizing covertness of the channel, 2 bit transfer mode should be used. For maximizing bandwidth, the raw data mode is the best. On networks with a small amount of loss, the 1 ACK mode works well, while on high loss networks the 2 ACK mode is optimal. The type of data sent may also dictate whether speed, reliability, or covertness is most desirable.

6. Experimental results

6.1 Initial Results

Initial experimentation has focused on manipulating the RTP timestamps and its resiliency to this manipulation. Based on initial experimentation, the timestamp can be manipulated within the expected range of 128 (7 bits) without any issues with receiving the voice stream. In further experimentation, it was determined that changes to the timestamp beyond the increment of 160 could be used, provided that the results are sequential. It is unclear whether this holds true for all phones, as it was only tested against a few phones available. Values of up to 256 were used with G.711, with no detectable degradation in call quality.

An example is shown in Figure 6 below, where the letter "A" (decimal 65) is embedded on the first packet of an RTP stream. In decimal, the total value is 225; 160 for the regular implement, plus 65 for the letter. During these transmissions, the audio reception was not impaired by this manipulation.

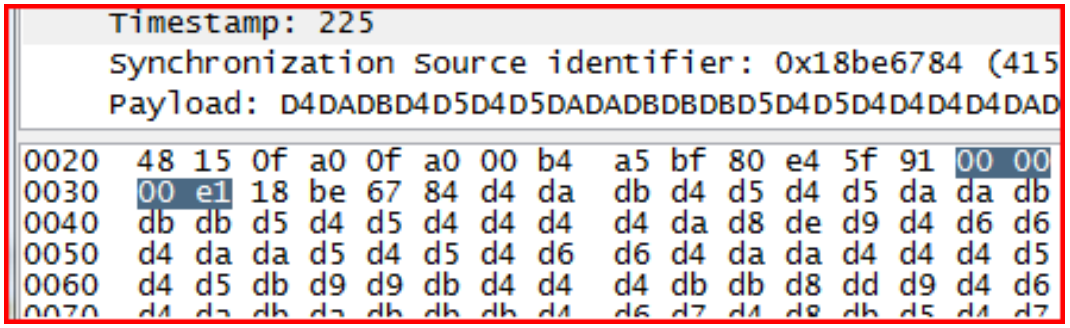


Figure 6: An ASCII "A" is embedded into the timestamp in the first RTP packet. The expected value of 160 has been replaced with 225.

Another experiment sought to add the new covert values to the timestamp without removing the previously used ones. This resulted in the timestamp increasingly growing out of sync with the real timestamp value. However, this also did not interrupt operations with the voice stream. This suggests that the timestamp values at which packets are dropped are based on the last received value rather than the overall expected timestamp.

6.2 Current Results

6.2.1 Speed

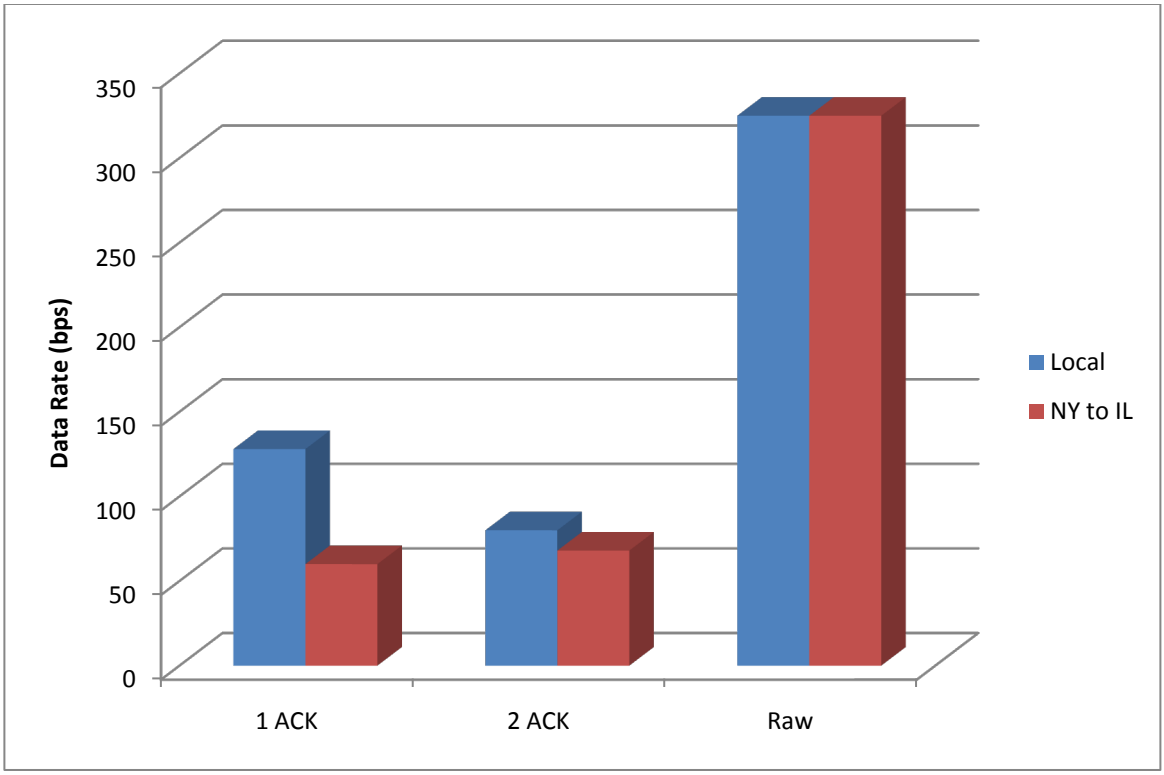


Figure 7: Speed of the data transfer with various modes of the reference implementation

Test	Speed Tx (bps)	Speed Rx (bps)
XP to Vista local – 1 ack	128	128
XP to Vista local – 2 ack	80	80
XP to Vista local - Raw	325	325
NY to IL – 1 ack	60	60
NY to IL – 2 ack	68	68
NY to IL - Raw	325	325

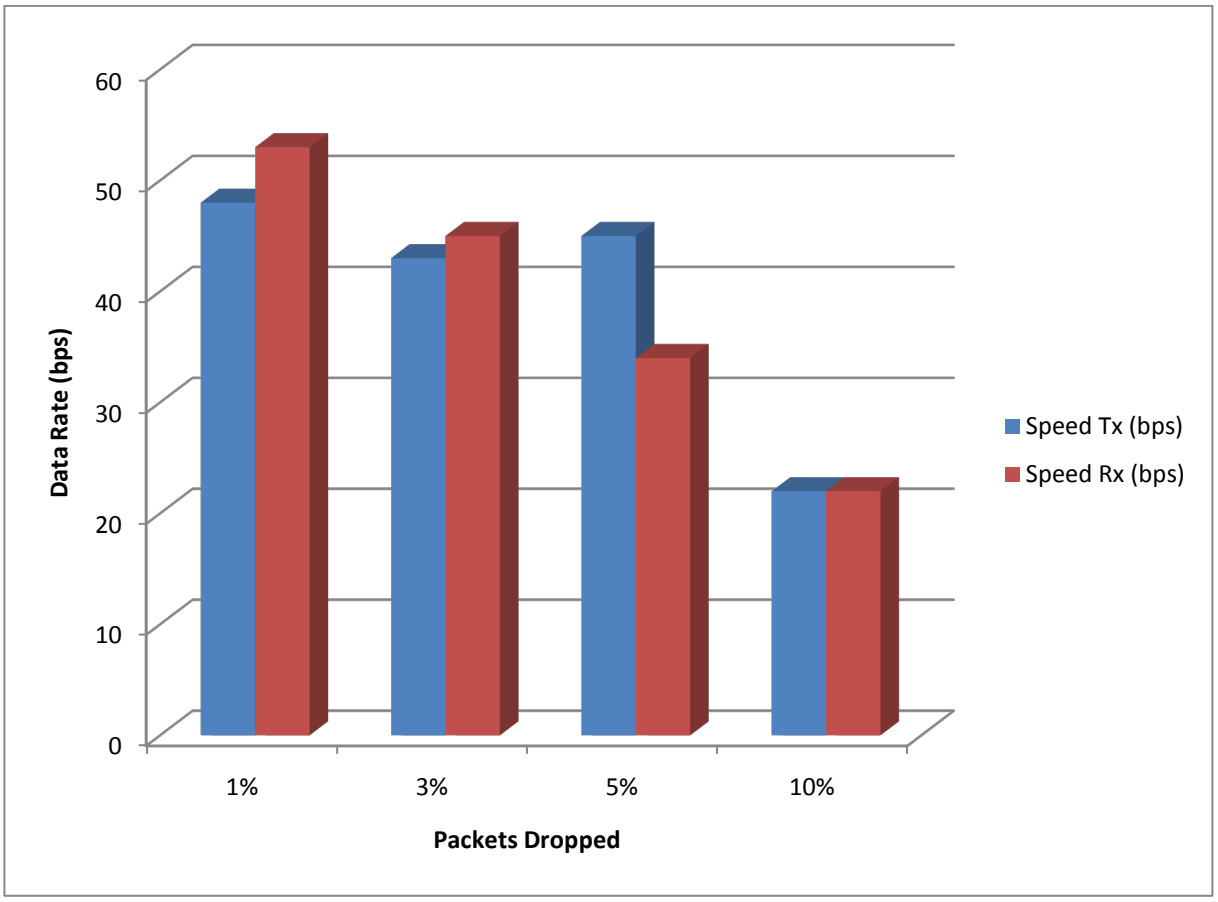


Figure 8: Speed of the data transfer using 2 ACKs at different amounts of packet loss

Percent Dropped (both directions)	Speed Tx (bps)	Speed Rx (bps)
1%	48	53
3%	43	45
5%	45	34
10%	22	22

The speed of the application when running in the raw mode (without reliability protocol) was close to the expected theoretical maximum. In tests, both local and from NY to IL, the speed transmission speed was 325 bps, compared to the theoretical maximum of 350 bps. It is not entirely clear what caused this slight discrepancy though it not surprising that it does not exactly match the theoretical rate. A minor change in timing of the packets could easily cause such a drop in rate.

When using the reliability protocol, the speed was also good, though it dropped notably from the raw data rate. When calling on a local network with the 2 ACK mode, speeds averaged 80 bps, while it dropped to 68 bps from NY to IL. The drop in speed as distance increases is expected using the ACK based systems, as round trip time increases and packets must complete a round trip before new data is sent. While this lower speed makes large data transfer impractical, small file transfer and two way text communications are easily accomplished.

6.2.2 Reliability

Test	Reliability Tx	Reliability Rx
XP to Vista local – 1 ack	Perfect	Perfect
XP to Vista local – 2 ack	Perfect	Perfect
XP to Vista local - Raw	Perfect	Perfect
NY to IL – 1 ack	Perfect	Perfect
NY to IL – 2 ack	Perfect	Perfect
NY to IL - Raw	Perfect	Perfect

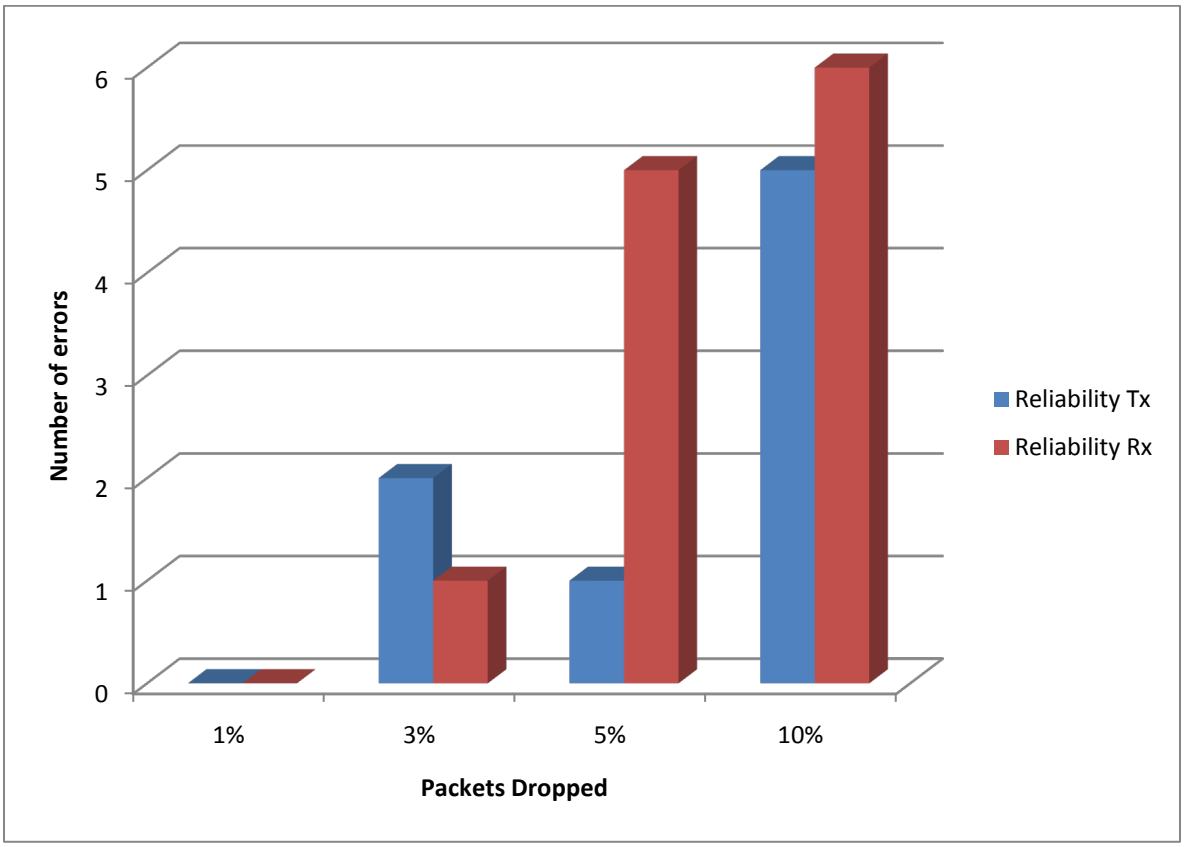


Figure 9: Errors in received data for different amounts of packet loss

Percent Dropped (both directions)	Reliability Tx	Reliability Rx
1%	Perfect	Perfect
3%	1024 to 1026 characters due to duplication	1024 to 1025 characters due to duplication
5%	1024 to 1025 characters due to duplication	1024 to 1029 characters due to duplication
10%	1024 to 1029 characters due to duplication	1024 to 1030 characters due to duplications from lost ACKs

In early tests, 100% accuracy for a 1k files was obtained about 90% of the time. When imperfections arose, it was duplicates of certain characters appearing due to lost ACKs. To minimize this from single packet loss, the dual ACK system was implemented, which improved performance. It was determined that the errors on the original local tests were due to running too many applications on the same computer rather than network loss, which at times delayed processing for the softphone. The least reliable setup was operating between a guest and host OS on the same machine over the VMware NAT network, due to the load on a single system. This proved to drop numerous packets at times, which could result in a duplicate appearing in a file (typically 1 in 1000 characters or less).

With the initial systems issues resolved, reliable transfers with no errors were repeatedly obtained over both the local network and across the public Internet. Raw mode transmitted flawlessly on all local and remote tests. It offered much faster data rates, with accuracy equal to the reliability mode. Even in the NY to IL tests, the data transfer was found to be accurate when using the raw client. The raw client also performed flawlessly on a local 300 KB test file. This persistence of the data indicates a quality Internet infrastructure in the northeastern US area and indicates that most calls could use the raw client as opposed to the more reliable one.

The client performed without error with up to 1% data loss, but experienced problems with 3% or higher loss. In those cases, duplicates were sometimes problematic. These results provided data that was easily readable for human communications, but could prove problematic the transferring of binary files. On a network experiencing a very high amount of data loss, files requiring perfect duplication should not be transmitted with this client.

6.2.3 Covertness

Overall the channel exhibited a high degree of covertness, making it difficult to detect. This was mainly due to the flexible nature of the RTP protocol and operation of the RTP timestamp. In the clients performing acknowledgments, the covert data was spread over time rather than being sequential, making it harder to detect. Acknowledgments also slow the rate at which data is sent, meaning many packets carry no data and use a default timestamp. This means that a data carrying packet needs to be located first, which increases the difficulty of detection. However, the acknowledgments themselves create additional network traffic, which could aid detection.

Attempts to view the covert channel proved difficult, even with knowledge of the covert channel. Even with a packet capture from Wireshark, it can be hard to see the covert data. Without knowledge of the channel, nothing appears unusual in the RTP packets. With knowledge of the channel, one must first find a data bearing packet, then determine the expected timestamp value, before being able to decode the data. Even if detected, the end product can be hard to decipher, as one has no means of knowing whether or not the receiving client successfully read the last piece of data. Without this knowledge, duplication could easily occur when attempting to decipher the stream.

The jitter experienced with the rapid rate of packets also leads to detection problems. Data on the wire may not arrive in the order it was sent, meaning the change in timestamps may be erratic on the wire. To effectively process this data while looking for the channel, one needs a jitter buffer and to arrange packets based on sequence numbers.

Firewalls inspecting only layers one through four are unable to block the channel; only application level firewalls with specific knowledge of the covert channel can detect it, and even then not entirely reliably. Applications firewalls properly configured for such a channel may potentially detect it, though they would face numerous challenges as detailed earlier.

Mathematically the channel is hard to detect since the RTP timestamp field need not follow any particular pattern. The RTP timestamp increment changes depending on protocol used for the RTP payload at that point in time, and may use system clock time. Additionally, the fact that the timestamp may not increment between packets further complicates analysis, and could make it easy to get false positives. In comparison, a payload based RTP covert channel may be detected by using mathematical steganographic analysis of payloads containing covert data.

Transfer Mode	Increment Entropy
Default	0
2 bit	0.079072
Raw	0.129782
1 ACK	0.188215
2 ACK	0.249292

Entropy of the timestamp increment for different transfer modes, given a G.711 VoIP implementation

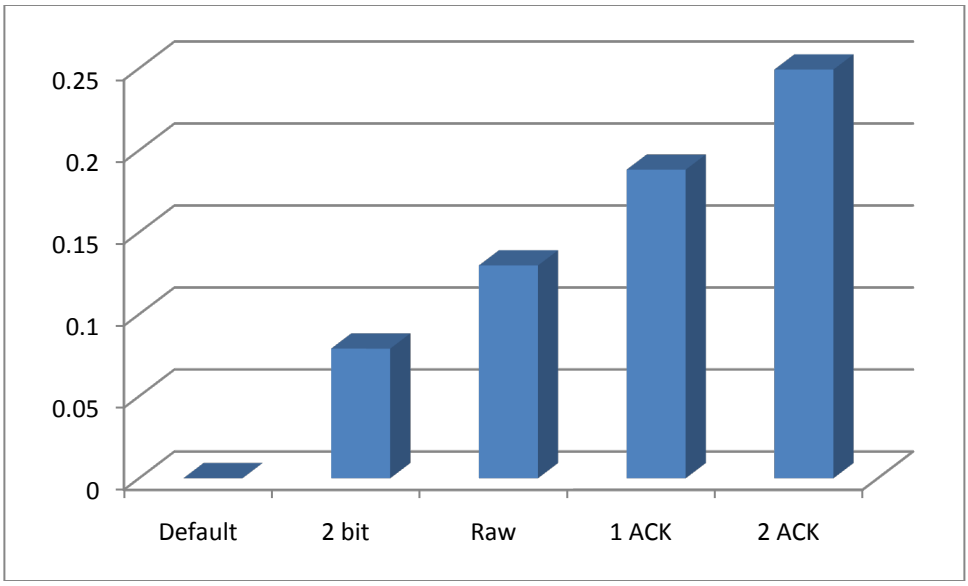


Figure 10: Entropy of the timestamp increment for G.711, for each of the different modes. Low bandwidth and low reliability is the most covert. Additional overhead from acknowledgments increases entropy. 1024 characters were sent on the covert channel out of 10,000 RTP packets for the above calculations.

However, it is possible to detect the covert channel provided knowledge of the codec used and particular implementation is available. This is done in one of two major ways: by looking for changes in the least significant bits of the timestamp, and by checking for entropy in the increment of the timestamp. While the former is fairly simple, the later is complicated by the flexible nature of RTP, the variety of codecs, and the varying nature of the timestamp. With knowledge of the expected timestamp value, one can detect variations in the incremental value. The entropy, relative to the timestamp increment, was calculated using the following formula, where p is the probability of the value observed in the tests and b is the base of the algorithm:

$$-\sum_{i=1}^{n} p(x_i) log_b p(x_i)$$

In general, the entropy of the channel increases as more variations are performed to the timestamp. To combat this, less data can be sent, and data can be sent over a longer period of time. While neither of these will reduce the entropy of the timestamp increment to zero, they can reduce it to less noticeable levels. However, the difficulties in determining the proper increment provide a layer of protection against such detection.

It is also difficult for an end user to detect the covert channel's presence. Detecting the running process is tricky since it is a small portion of code integrated into the phone client. End user detection by call operation or quality is also unlikely based on test results. Out of five people rating call quality, none perceived any difference in call quality when the covert data was added.

7. Future work

Future work will focus on further developing the reference implementation and strengthening the design of this covert channel. The focus will be on attempting to ensure reliability while maintaining the highest possible data rate and lowest risk of detection. Since these are somewhat competing objectives, a balanced medium needs to be established.

A standalone version of this covert channel is also planned for later development. The standalone version will allow for use with hardware phones, as well as with any soft client. In addition, the standalone implementation will be able to perform man in the middle style communications on calls made by others. The idea for this client is flexibility in use, allowing it to be used from a call endpoint or as an inline proxy.

To help hide the data transmission, the traffic may be obfuscated by simple XOR operation against a pre shared key. This increases entropy of the transmitted data, making it more difficult to decipher. This also provides a weak way of indentifying a known sender, though not any particular sender and should not be relied upon. A preamble may also be implemented at the beginning of a communications to indicate the sender.

A graphical user interface will also be added to the reference implementation to provide for ease of use and to eliminate errors. This interface will include the ability to transmit text and files across the channel and to view the results received. Two boxes could be implemented, one for input and one for output. In addition, files could be selected to be used for the input or

output. Other options to tweak operations, such as enabling or disabling reliability mode, could also be implemented.

The future outlook is good for this covert channel, as there is a growing trend towards increasing sampling rates to 16 kHz from 8 kHz, allowing for values up to 320 to be easily transmitted as opposed to 160. This increased quality reflects better bandwidth availability and low processing costs with modern hardware. In addition, VoIP and other RTP users are only set to continue to grow over time, increasing opportunities for usage almost anywhere at any time.

Future research could also be conducted on the use of the additional data stream provided by the covert channel for more overt usages. For example, phones could be configured to exchange additional information during a phone call with no additional packet creation or bandwidth used. Data similar to that contained in RTCP packets could be embedded into the RTP itself. The key benefit in this scenario is the bandwidth saving, and would likely be used with protocols such as G.729 where attempts to conserve bandwidth are already in use. The additional transmitted data could also provide diagnostic information, perhaps similar to RTCP without the additional associated network traffic.

8. Conclusion

We have shown the ability to transmit and receive data using a new covert channel over RTP, without interrupting the reception of the voice stream. This reference implementation shows that this covert channel can be practically implemented and used. Speed was shown to be sufficient for two-way text based communications and small file transfer. Reliability was also shown to be good, if not always perfect. The ability to detect this channel was also seen to be difficult, given the many packets involved and the way RTP operates. Future work will further develop this reference implementation with increased reliability, flexibility, and usability.

References

[1] Druid (2007, September). "Real-time Steganography with RTP". Uniformed [Online]. Vol. 8. Available http://www.uninformed.org/?v=8&a=3&t=sumry.

[2] Takahashi, Takehiro; Lee, Wenke, "An assessment of VoIP covert channel threats," Security and Privacy in Communications Networks and the Workshops, 2007. SecureComm 2007. p. 371-380, 17-21 Sept. 2007.

[3] Hui Tian, Ke Zhou, Hong Jiang, Yongfeng Huang, Jin Liu, and Dan Feng, "An M-Sequence Based Steganography Model for Voice over IP," accepted to appear in the Proceedings of the 2009 IEEE International Conference on Communications (ICC'09), Dresden, Germany, June 14-18, 2009.

[4] W. Mazurczyk, Z. Kotulski, "Covert channel for improving VoIP security". in: J.Pejaś, Kh. Saeed [Eds], Advances in Information Processing and Protection, pp. 271-280, Springer, Berlin 2007. ISBN: 978-0-387-73136-0, 978-0-387-73137-7.

[5] W. Mazurczyk, J. Lubacz, K. Szczypiorski, "Hiding Data in VoIP", In Proc of: The 26th Army Science Conference (ASC 2008), Orlando, Florida, USA, December 1-4, 2008.

[6] Wang, X., Chen, S., and Jajodia, S. 2005. "Tracking anonymous peer-to-peer VoIP calls on the internet". In Proceedings of the 12th ACM Conference on Computer and Communications Security (Alexandria, VA, USA, November 07 - 11, 2005). CCS '05. ACM, New York, NY, 81-91. DOI= http://doi.acm.org/10.1145/1102120.1102133.

[7] John Griffin, Rachel Greenstadt, Peter Litwack, Richard Tibbetts. "Covert Messaging Through TCP Timestamps", Massachusetts Institute of Technology.

[8] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 3550, July 2003.

[9] H. Schulzrinne, "RTP: Some Frequently Asked Questions about RTP," 2008, http://www.cs.columbia.edu/~hgs/rtp/faq.html.

[10] H. Schulzrinne and S. Casner, "RTP Profile for Audio and Video Conferences with Minimal Control", RFC 3551, July 2003.

[11] H.Schulzrinne, “Audio Codecs,” 2008,
http://www1.cs.columbia.edu/~hgs/audio/codecs.html.

[12] Jean-Marc Valin, “Programming with Speex (the libspeex API),” 2007,
http://www.speex.org/docs/manual/speex-manual/node7.html.

[13] Xiph.org, “Speex: Codec Quality Comparison,” 2006, http://speex.org/comparison/.

[14] Internet Assigned Numbers Authority, “Real-Time Transport Protocol (RTP) Parameters,” 2009, http://www.iana.org/assignments/rtp-parameters.

[15] “Pulse Code Modulation(PCM) of Voice Frequencies”, ITU-T Recommendation G.711. International Telecommunication Union, Telecommunication Union (ITU-T). 1993. Available http://www.itu.int/rec/T-REC-G.711-198811-I/en.

[16] Gerald Combs. Wireshark Network Protocol Analyzer, version 1.07, 2009. Available http://www.wireshark.org/.

[17] Microsoft Windows SDK for Windows Server 2008 and .NET Framework 3.5. Microsoft Corporation, 2008. Available http://msdn.microsoft.com/en-us/windows/bb980924.aspx.

[18] Microsoft DirectX SDK (March 2009). Microsoft Corporation, 2009. Available http://msdn.microsoft.com/directx/.

[19] Prijono, Benny. pjproject-1.3. 2009. Available http://www.pjsip.org/download.htm.

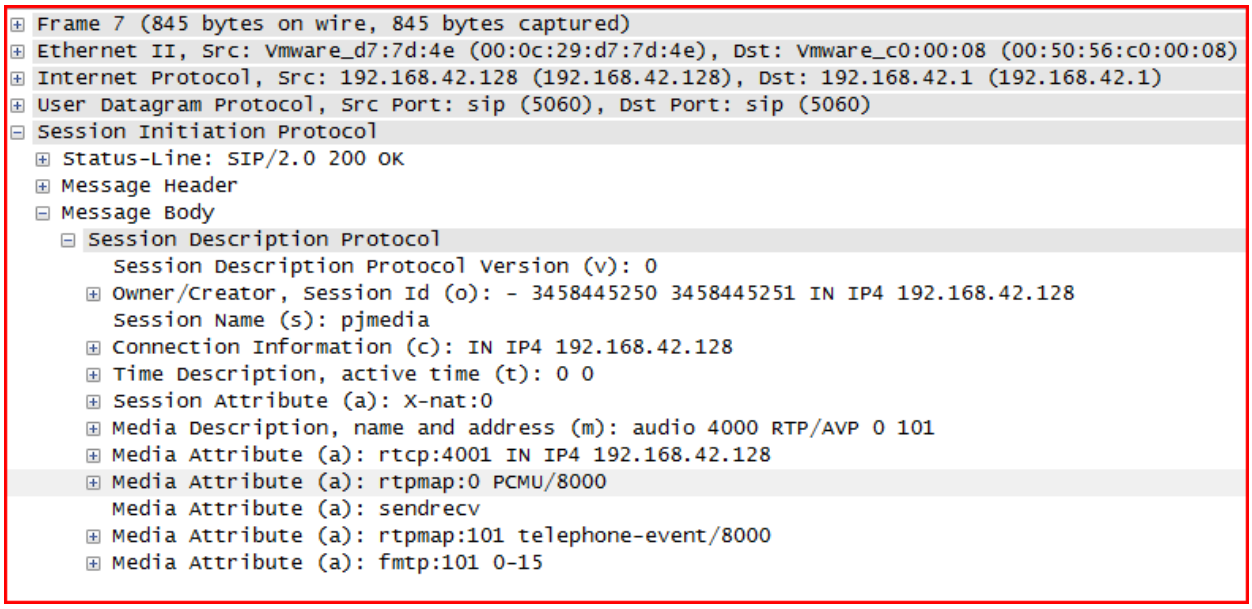
[20] AsteriskNOW 1.5.0. Digium, Inc, 2009. Available http://www.asterisknow.org/.

[21] “Media Flow – PJSIP,” http://trac.pjsip.org/repos/wiki/media-flow.

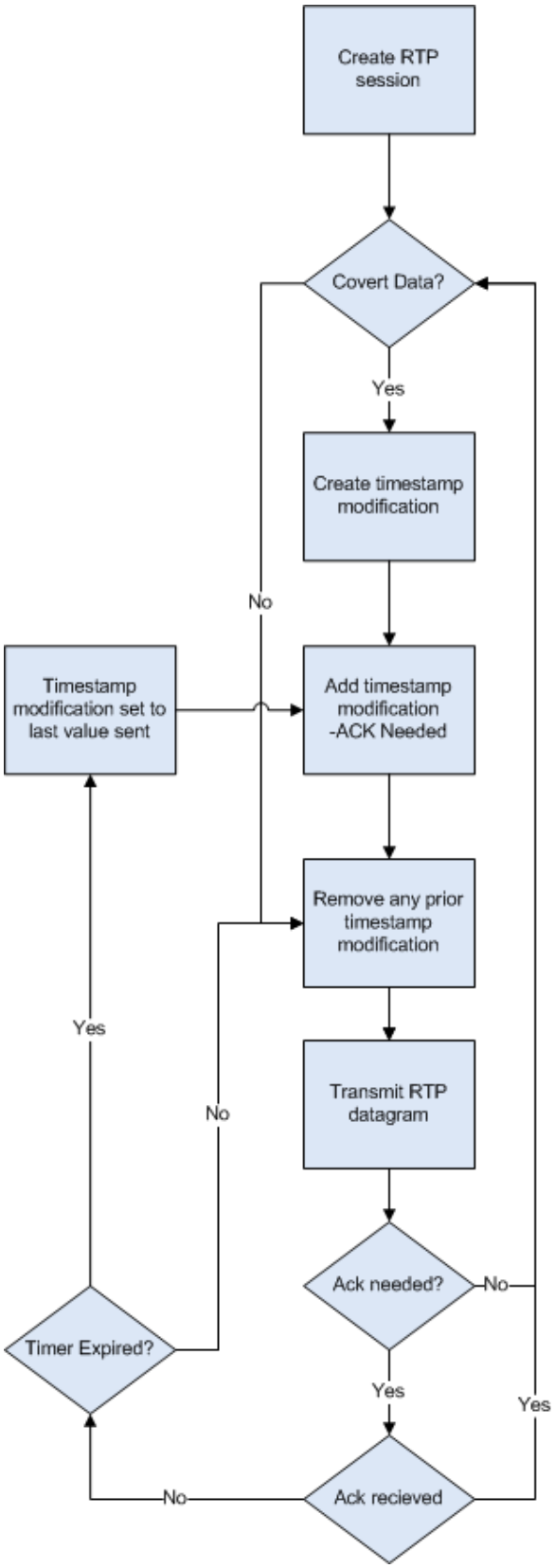
[22] X-Lite, version 3.0 build 53117. ConterPath Corporation, 2009. Available http://www.counterpath.net/x-lite.html.

Appendices

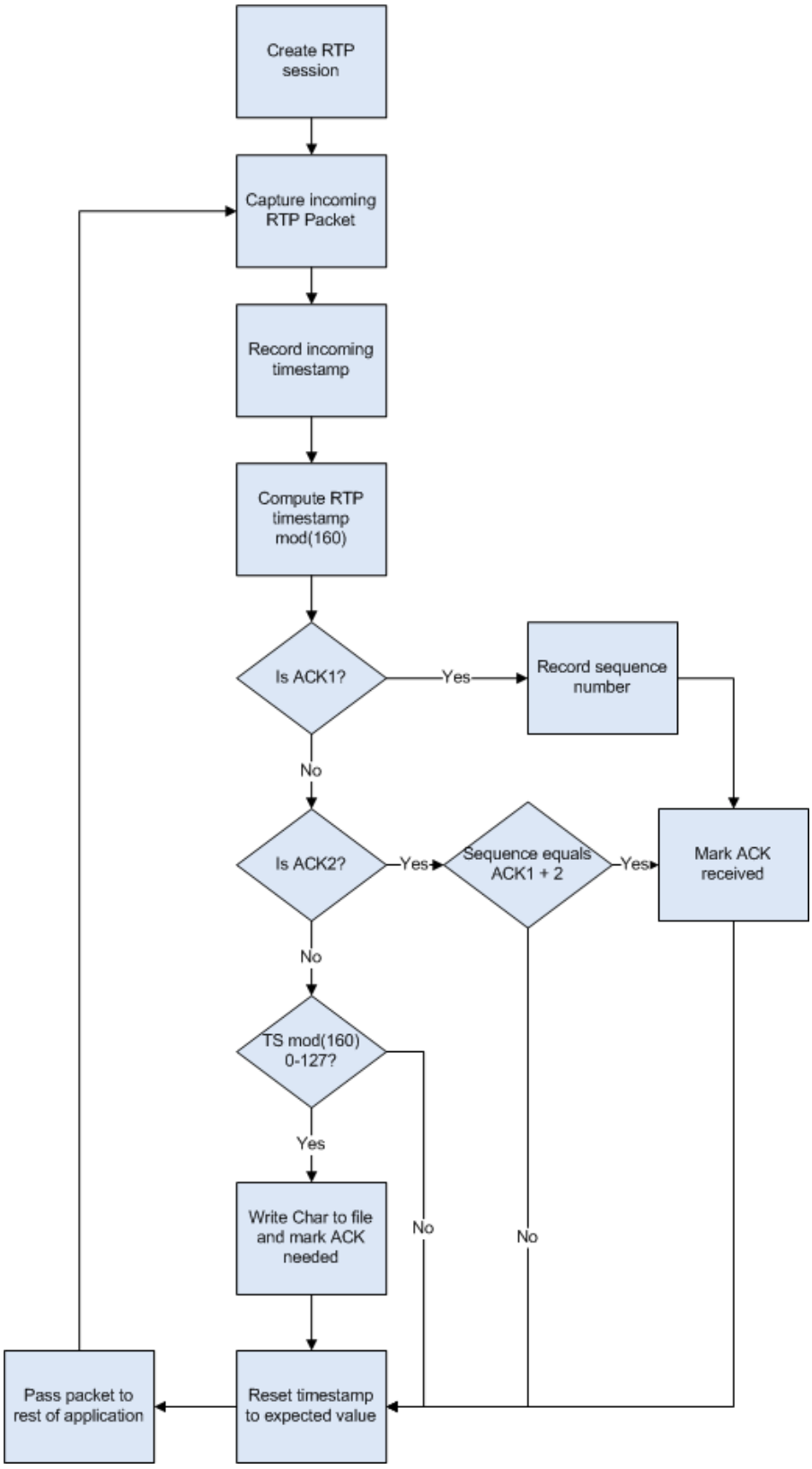
Appendix 1: SIP/SDP Call Setup



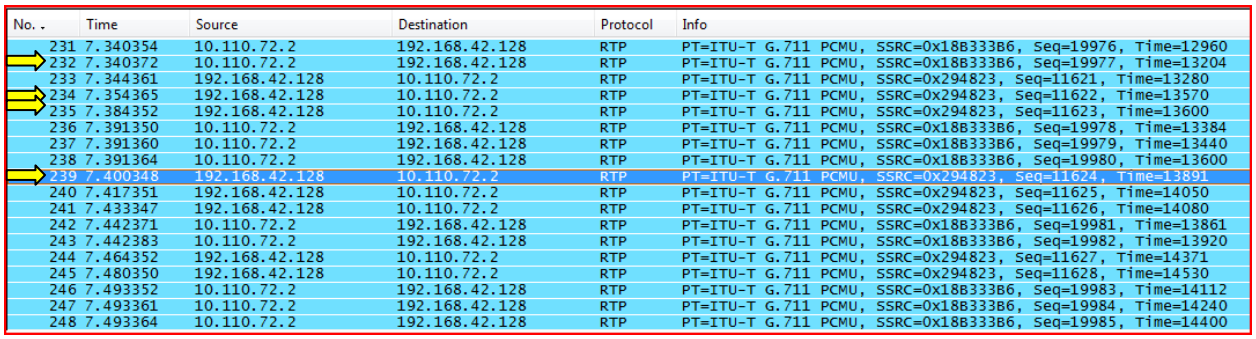
Appendix 2: Transmit Flow Chart



Appendix 3: Receive Flow Chart



Appendix 4: Dual acknowledgment packets



No.	Time	Source	Destination	Protocol	Info
231	7.340354	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19976, Time=12960
232	7.340372	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19977, Time=13204
233	7.344361	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11621, Time=13280
234	7.354365	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11622, Time=13570
235	7.384352	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11623, Time=13600
236	7.391350	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19978, Time=13384
237	7.391360	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19979, Time=13440
238	7.391364	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19980, Time=13600
239	7.400348	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11624, Time=13891
240	7.417351	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11625, Time=14050
241	7.433347	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11626, Time=14080
242	7.442371	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19981, Time=13861
243	7.442383	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19982, Time=13920
244	7.464352	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11627, Time=14371
245	7.480350	192.168.42.128	10.110.72.2	RTP	PT=ITU-T G.711 PCMU, SSRC=0x294823, Seq=11628, Time=14530
246	7.493352	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19983, Time=14112
247	7.493361	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19984, Time=14240
248	7.493364	10.110.72.2	192.168.42.128	RTP	PT=ITU-T G.711 PCMU, SSRC=0x18B333B6, Seq=19985, Time=14400

Packet 232: 10.110.72.2 sends the value “84” (the letter “T”)

Packet 234: 192.168.42.128 sends first acknowledgment

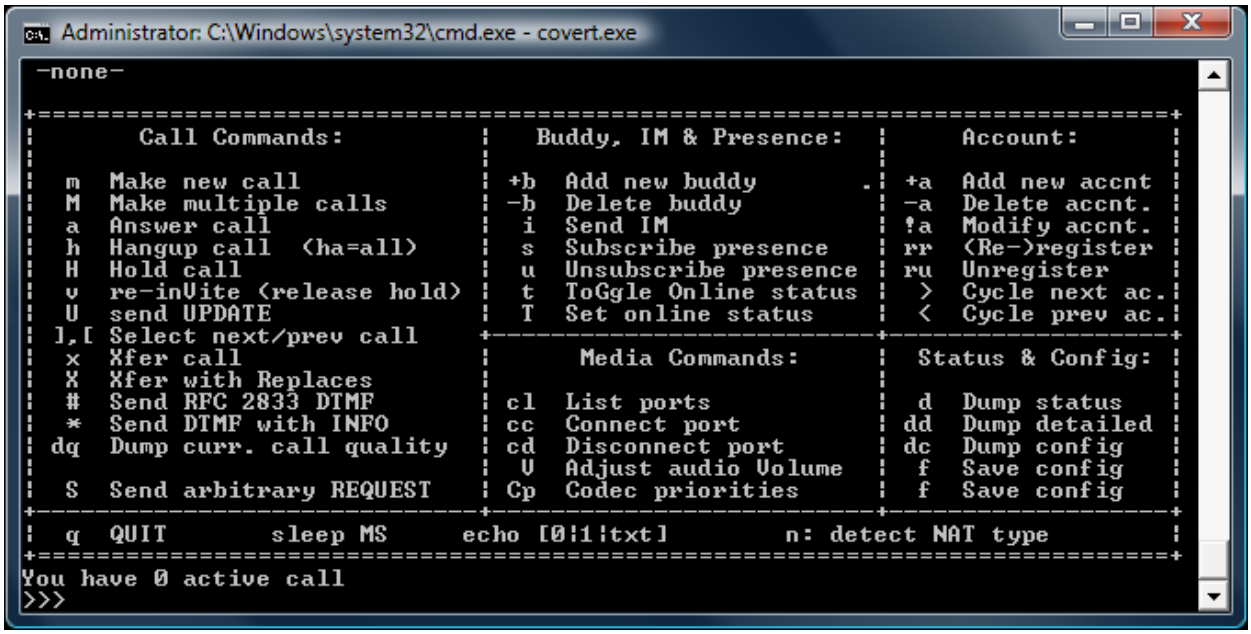
Packet 235: 192.168.42.128 sends normal timestamp

Packet 239: 192.168.42.128 sends seconds acknowledgment

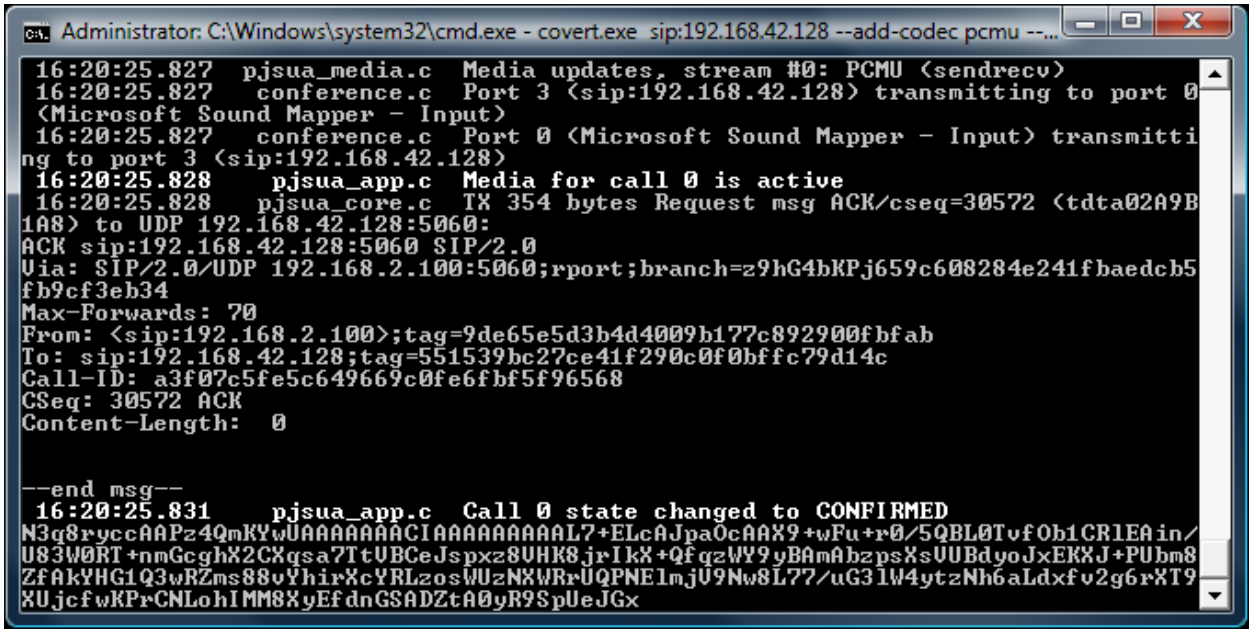
Appendix 5: NY to IL Call Statistics

```
#0 PCMU @8KHz, sendrecv, peer=98.228.90.198:4000
   RX pt=0, stat last update: 00h:00m:01.605s ago
      total 1.8Kpkt 294.4KB (368.0KB +IP hdr) @avg=63.2Kbps/79.0Kbps
      pkt loss=0 (0.0%), discrd=0 (0.0%), dup=0 (0.0%), reord=0 (0.0%)
            (msec)    min     avg     max     last    dev
      loss period:   0.000   0.000   0.000   0.000   0.000
      jitter     :   0.000   2.361  49.125   1.125   3.402
   TX pt=0, ptime=20ms, stat last update: 00h:00m:01.557s ago
      total 1.8Kpkt 297.2KB (371.6KB +IP hdr) @avg 63.8Kbps/79.8Kbps
      pkt loss=2 (0.1%), dup=0 (0.0%), reorder=0 (0.0%)
            (msec)    min     avg     max     last    dev
      loss period:  20.000  20.000  20.000  20.000   0.000
      jitter     :   0.000  19.688  26.000  22.250   7.696
  RTT msec       :  58.425  87.409 151.000 143.000  61.722
```

Appendix 6: Reference Implementation



Main Screen of application, using PJSUA



Covert data being transferred while call in progress

Appendix 7: rtp.c source code (acknowledging)

```
/* $Id: rtp.c 2394 2008-12-23 17:27:53Z bennylp $ */
/*
 * Copyright (C) 2008-2009 Teluu Inc. (http://www.teluu.com)
 * Copyright (C) 2003-2008 Benny Prijono <benny@prijono.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
#include <pjmedia/rtp.h>
#include <pjmedia/errno.h>
#include <pj/log.h>
#include <pj/sock.h>    /* pj_htonx, pj_htonx */
#include <pj/assert.h>
#include <pj/rand.h>
#include <pj/string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define THIS_FILE   "rtp.c"

#define RTP_VERSION     2

#define RTP_SEQ_MOD     (1 << 16)
#define MAX_DROPOUT     ((pj_int16_t)3000)
#define MAX_MISORDER    ((pj_int16_t)100)
#define MIN_SEQUENTIAL  ((pj_int16_t)2)
#define IN_FILE                 "input.txt"
#define OUT_FILE        "output.txt"
#define DELAY_TIME              2
#define SEND_DELAY              1
FILE *infp;       //input file
FILE *outfp;      //output file
time_t start_time;
time_t cur_time;
time_t call_start;


static void pjmedia_rtp_seq_restart(pjmedia_rtp_seq_session *seq_ctrl,
                                    pj_uint16_t seq);
```

```
PJ_DEF(pj_status_t) pjmedia_rtp_session_init( pjmedia_rtp_session *ses,
                                      int default_pt,
                                      pj_uint32_t sender_ssrc )
{
    PJ_LOG(5, (THIS_FILE,
              "pjmedia_rtp_session_init: ses=%p, default_pt=%d, ssrc=0x%x",
              ses, default_pt, sender_ssrc));

    /* Check RTP header packing. */
    if (sizeof(struct pjmedia_rtp_hdr) != 12) {
      pj_assert(!"Wrong RTP header packing!");
      return PJMEDIA_RTP_EINPACK;
    }

    /* If sender_ssrc is not specified, create from random value. */
    if (sender_ssrc == 0 || sender_ssrc == (pj_uint32_t)-1) {
      sender_ssrc = pj_htonl(pj_rand());
    } else {
      sender_ssrc = pj_htonl(sender_ssrc);
    }

    /* Initialize session. */
    pj_bzero(ses, sizeof(*ses));

    /* Initial sequence number SHOULD be random, according to RFC 3550. */
    /* According to RFC 3711, it should be random within 2^15 bit */
    ses->out_extseq = pj_rand() & 0x7FFF;
    ses->peer_ssrc = 0;

    /* Build default header for outgoing RTP packet. */
    ses->out_hdr.v = RTP_VERSION;
    ses->out_hdr.p = 0;
    ses->out_hdr.x = 0;
    ses->out_hdr.cc = 0;
    ses->out_hdr.m = 0;
    ses->out_hdr.pt = (pj_uint8_t) default_pt;
    ses->out_hdr.seq = (pj_uint16_t) pj_htons( (pj_uint16_t)ses->out_extseq
);
    ses->out_hdr.ts = 0;
    ses->out_hdr.ssrc = sender_ssrc;

    /* Keep some arguments as session defaults. */
    ses->out_pt = (pj_uint16_t) default_pt;

      /* open files to read write */
      infp = fopen(IN_FILE, "r");
      outfp = fopen(OUT_FILE, "w+");

      //initalize covert values
      ack_rcvd=1;
      ts_mod_last=0;
      ack_send=0;
      ready_send=0;
      time(&call_start);
      retrans_rcv = 0;
      retrans_send = 0;
      ack2_send = 0;
```

```
    return PJ_SUCCESS;
}

PJ_DEF(pj_status_t) pjmedia_rtp_session_init2(
                           pjmedia_rtp_session *ses,
                           pjmedia_rtp_session_setting settings)
{
    pj_status_t status;
    int            pt = 0;
    pj_uint32_t    sender_ssrc = 0;

    if (settings.flags & 1)
      pt = settings.default_pt;
    if (settings.flags & 2)
      sender_ssrc = settings.sender_ssrc;

    status = pjmedia_rtp_session_init(ses, pt, sender_ssrc);
    if (status != PJ_SUCCESS)
      return status;

    if (settings.flags & 4) {
      ses->out_extseq = settings.seq;
      ses->out_hdr.seq = pj_htons((pj_uint16_t)ses->out_extseq);
    }
    if (settings.flags & 8)
      ses->out_hdr.ts = pj_htonl(settings.ts);

    return PJ_SUCCESS;
}


PJ_DEF(pj_status_t) pjmedia_rtp_encode_rtp( pjmedia_rtp_session *ses,
                                    int pt, int m,
                                    int payload_len, int ts_len,
                                    const void **rtphdr, int *hdrlen )
{
    PJ_UNUSED_ARG(payload_len);

      //set prior ts modification value
      if(ts_mod_old<160 && ts_mod_old>=(-1)){
      }
      else{
            ts_mod_old=0;
      }

      //set ts_mod to zero in case no new data
      ts_mod=0;

      time(&cur_time);

      //delay mechanism if needed at start.  disabled if readysend = (1)
      if(ready_send==0){
            if(difftime(cur_time, call_start)>SEND_DELAY){
                  ready_send=1;
            }
```

```
	}
	//take a pause from first ack
	else if(ts_mod_old==130){
		ts_mod=0;
		ack2_send=1;
	}
	//send second ack if first sent
	else if (ack2_send==1){
		ts_mod=131;
		ack2_send=0;
	}
	//check if need to send an ack
	else if(ack_send == 1){
		ts_mod=130;
		ack_send=0;
	}
	//check time since last transmit if no ack rcvd
	else if((difftime(cur_time, start_time) >= DELAY_TIME) &&
(ack_rcvd==0)){
		ts_mod=ts_mod_last;

		//reset timer start
		time(&start_time);
	}
	//transmit new data if last recieved
	else if(ack_rcvd == 1){
		//set timestamp modification for current packet
		ts_mod=(getc(infp));

		//reset null to 128
		if (ts_mod == 0) {
			ts_mod=128;
		}

		//adjust if EOF. no ack for
		if (ts_mod == -1){
			ts_mod=0;
			ack_rcvd=1;
		}
		else if (ts_mod == 130){
			ack_rcvd=1;
		}
		else{
			//reset ack value
			ack_rcvd=0;

			//set timer start
			time(&start_time);

			//set transmitted value to last sent
			ts_mod_last = ts_mod;
		}
	}


    /* Update timestamp */
```

```
    ses->out_hdr.ts = pj_htonl(pj_ntohl(ses->out_hdr.ts)+ts_len+ts_mod-
ts_mod_old);

      //undo timestamp offset
      ts_mod_old=ts_mod;



    /* If payload_len is zero, bail out.
     * This is a clock frame; we're not really transmitting anything.
     */
    //if (payload_len == 0)
      //return PJ_SUCCESS;

    /* Update session. */
    ses->out_extseq++;

    /* Create outgoing header. */
    ses->out_hdr.pt = (pj_uint8_t) ((pt == -1) ? ses->out_pt : pt);
    ses->out_hdr.m = (pj_uint16_t) m;
    ses->out_hdr.seq = pj_htons( (pj_uint16_t) ses->out_extseq);

    /* Return values */
    *rtphdr = &ses->out_hdr;
    *hdrlen = sizeof(pjmedia_rtp_hdr);

    return PJ_SUCCESS;
}


PJ_DEF(pj_status_t) pjmedia_rtp_decode_rtp( pjmedia_rtp_session *ses,
                                    const void *pkt, int pkt_len,
                                    pjmedia_rtp_hdr **hdr,
                                    const void **payload,
                                    unsigned *payloadlen)
{
    int offset;
      pj_uint32_t ts_diff;
      pj_uint32_t ts_tmp;

    PJ_UNUSED_ARG(ses);

    /* Assume RTP header at the start of packet. We'll verify this later. */
    *hdr = (pjmedia_rtp_hdr*)pkt;

    /* Check RTP header sanity. */
    if ((*hdr)->v != RTP_VERSION) {
      return PJMEDIA_RTP_EINVER;
    }

      /*decode covert data*/
      ts_tmp = pj_ntohl((*hdr)->ts);
      ts_diff=( ts_tmp % 160);
      //record ACK 1 Recieved
      if(ts_diff == 130){
            ack_rcvd=1;
            ack_seq=pj_ntohs((*hdr)->seq);
```

```
            //printf("%d ",ack_seq);
      }
      //record ACK2 recieved in ACK one is not recieved
      else if((ts_diff == 131) && (ack_rcvd==0) && (pj_ntohs((*hdr)-
>seq)!=(ack_seq+2))){
            ack_rcvd=1;
            //putchar(33);
      }
      else if ((ts_diff < 129) && (ts_diff > 0)){
            if(ts_diff == 128){
                  ts_diff=0;
            }
            putchar(ts_diff);
            fputc(ts_diff,outfp);
            ack_send=1;
      }
      //adjust timestamp to expected value for program to process
      (*hdr)->ts = pj_htonl(ts_tmp-ts_diff);
      //printf("%d",pj_ntohl((*hdr)->ts));

    /* Payload is located right after header plus CSRC */
    offset = sizeof(pjmedia_rtp_hdr) + ((*hdr)->cc * sizeof(pj_uint32_t));

    /* Adjust offset if RTP extension is used. */
    if ((*hdr)->x) {
      pjmedia_rtp_ext_hdr *ext = (pjmedia_rtp_ext_hdr*)
                           (((pj_uint8_t*)pkt) + offset);
      offset += (pj_ntohs(ext->length) * sizeof(pj_uint32_t));
    }

    /* Check that offset is less than packet size */
    if (offset > pkt_len)
      return PJMEDIA_RTP_EINLEN;

    /* Find and set payload. */
    *payload = ((pj_uint8_t*)pkt) + offset;
    *payloadlen = pkt_len - offset;

    return PJ_SUCCESS;
}


PJ_DEF(void) pjmedia_rtp_session_update( pjmedia_rtp_session *ses,
                                const pjmedia_rtp_hdr *hdr,
                                pjmedia_rtp_status *p_seq_st)
{
    pjmedia_rtp_session_update2(ses, hdr, p_seq_st, PJ_TRUE);
}

PJ_DEF(void) pjmedia_rtp_session_update2( pjmedia_rtp_session *ses,
                                 const pjmedia_rtp_hdr *hdr,
                                 pjmedia_rtp_status *p_seq_st,
                                 pj_bool_t check_pt)
{
    pjmedia_rtp_status seq_st;

    /* for now check_pt MUST be either PJ_TRUE or PJ_FALSE.
```

```
     * In the future we might change check_pt from boolean to
     * unsigned integer to accommodate more flags.
     */
    pj_assert(check_pt==PJ_TRUE || check_pt==PJ_FALSE);

    /* Init status */
    seq_st.status.value = 0;
    seq_st.diff = 0;

    /* Check SSRC. */
    if (ses->peer_ssrc == 0) ses->peer_ssrc = pj_ntohl(hdr->ssrc);

    if (pj_ntohl(hdr->ssrc) != ses->peer_ssrc) {
      seq_st.status.flag.badssrc = 1;
      ses->peer_ssrc = pj_ntohl(hdr->ssrc);
    }

    /* Check payload type. */
    if (check_pt && hdr->pt != ses->out_pt) {
      if (p_seq_st) {
          p_seq_st->status.value = seq_st.status.value;
          p_seq_st->status.flag.bad = 1;
          p_seq_st->status.flag.badpt = 1;
      }
      return;
    }

    /* Initialize sequence number on first packet received. */
    if (ses->received == 0)
      pjmedia_rtp_seq_init( &ses->seq_ctrl, pj_ntohs(hdr->seq) );

    /* Check sequence number to see if remote session has been restarted. */
    pjmedia_rtp_seq_update( &ses->seq_ctrl, pj_ntohs(hdr->seq), &seq_st);
    if (seq_st.status.flag.restart) {
      ++ses->received;

    } else if (!seq_st.status.flag.bad) {
      ++ses->received;
    }

    if (p_seq_st) {
      p_seq_st->status.value = seq_st.status.value;
      p_seq_st->diff = seq_st.diff;
    }
}


void pjmedia_rtp_seq_restart(pjmedia_rtp_seq_session *sess, pj_uint16_t seq)
{
    sess->base_seq = seq;
    sess->max_seq = seq;
    sess->bad_seq = RTP_SEQ_MOD + 1;
    sess->cycles = 0;
}
```

```
void pjmedia_rtp_seq_init(pjmedia_rtp_seq_session *sess, pj_uint16_t seq)
{
    pjmedia_rtp_seq_restart(sess, seq);

    sess->max_seq = (pj_uint16_t) (seq - 1);
    sess->probation = MIN_SEQUENTIAL;
}


void pjmedia_rtp_seq_update( pjmedia_rtp_seq_session *sess,
                        pj_uint16_t seq,
                        pjmedia_rtp_status *seq_status)
{
    pj_uint16_t udelta = (pj_uint16_t) (seq - sess->max_seq);
    pjmedia_rtp_status st;

    /* Init status */
    st.status.value = 0;
    st.diff = 0;

    /*
     * Source is not valid until MIN_SEQUENTIAL packets with
     * sequential sequence numbers have been received.
     */
    if (sess->probation) {

      st.status.flag.probation = 1;

        if (seq == sess->max_seq+ 1) {
          /* packet is in sequence */
          st.diff = 1;
          sess->probation--;
            sess->max_seq = seq;
            if (sess->probation == 0) {
            st.status.flag.probation = 0;
            }
      } else {

          st.diff = 0;

          st.status.flag.bad = 1;
          if (seq == sess->max_seq)
            st.status.flag.dup = 1;
          else
            st.status.flag.outorder = 1;

          sess->probation = MIN_SEQUENTIAL - 1;
          sess->max_seq = seq;
        }


    } else if (udelta == 0) {

      st.status.flag.dup = 1;

    } else if (udelta < MAX_DROPOUT) {
      /* in order, with permissible gap */
```

```
      if (seq < sess->max_seq) {
          /* Sequence number wrapped - count another 64K cycle. */
          sess->cycles += RTP_SEQ_MOD;
        }
        sess->max_seq = seq;

      st.diff = udelta;

    } else if (udelta <= (RTP_SEQ_MOD - MAX_MISORDER)) {
      /* the sequence number made a very large jump */
        if (seq == sess->bad_seq) {
          /*
           * Two sequential packets -- assume that the other side
           * restarted without telling us so just re-sync
           * (i.e., pretend this was the first packet).
           */
          pjmedia_rtp_seq_restart(sess, seq);
          st.status.flag.restart = 1;
          st.status.flag.probation = 1;
          st.diff = 1;
      }
        else {
          sess->bad_seq = (seq + 1) & (RTP_SEQ_MOD-1);
            st.status.flag.bad = 1;
          st.status.flag.outorder = 1;
        }
    } else {
      /* old duplicate or reordered packet.
       * Not necessarily bad packet (?)
       */
      st.status.flag.outorder = 1;
    }


    if (seq_status) {
      seq_status->diff = st.diff;
      seq_status->status.value = st.status.value;
    }
}
```

Appendix 8: rtp.h source code

```
/* $Id: rtp.h 2394 2008-12-23 17:27:53Z bennylp $ */
/*
 * Copyright (C) 2008-2009 Teluu Inc. (http://www.teluu.com)
 * Copyright (C) 2003-2008 Benny Prijono <benny@prijono.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
#ifndef __PJMEDIA_RTP_H__
#define __PJMEDIA_RTP_H__


/**
 * @file rtp.h
 * @brief RTP packet and RTP session declarations.
 */
#include <pjmedia/types.h>


PJ_BEGIN_DECL


/**
 * @defgroup PJMED_RTP RTP Session and Encapsulation (RFC 3550)
 * @ingroup PJMEDIA_SESSION
 * @brief RTP format and session management
 * @{
 *
 * The RTP module is designed to be dependent only to PJLIB, it does not
depend
 * on any other parts of PJMEDIA library. The RTP module does not even depend
 * on any transports (sockets), to promote even more use, such as in DSP
 * development (where transport may be handled by different processor).
 *
 * An RTCP implementation is available, in separate module. Please see
 * @ref PJMED_RTCP.
 *
 * The functions that are provided by this module:
 *  - creating RTP header for each outgoing packet.
 *  - decoding RTP packet into RTP header and payload.
 *  - provide simple RTP session management (sequence number, etc.)
 *
 * The RTP module does not use any dynamic memory at all.
```

```
 *
 * \section P1 How to Use the RTP Module
 *
 * First application must call #pjmedia_rtp_session_init() to initialize the
RTP
 * session.
 *
 * When application wants to send RTP packet, it needs to call
 * #pjmedia_rtp_encode_rtp() to build the RTP header. Note that this WILL NOT
build
 * the complete RTP packet, but instead only the header. Application can
 * then either concatenate the header with the payload, or send the two
 * fragments (the header and the payload) using scatter-gather transport API
 * (e.g. \a sendv()).
 *
 * When application receives an RTP packet, first it should call
 * #pjmedia_rtp_decode_rtp to decode RTP header and payload, then it should
call
 * #pjmedia_rtp_session_update to check whether we can process the RTP
payload,
 * and to let the RTP session updates its internal status. The decode
function
 * is guaranteed to point the payload to the correct position regardless of
 * any options present in the RTP packet.
 *
 */

#ifdef _MSC_VER
#   pragma warning(disable:4214)    // bit field types other than int
#endif

 /**
 * set values for covert channel
 * old timestamp mod value
 * mod value
 * last mod value sent (for retrans)
 * need to send ack
 * ack was recieved for last
 *
 */
int ts_mod_old;
int ts_mod;
int ts_mod_last;
int ack_send;
int ack_rcvd;
int ready_send;
int retrans_rcv;
int retrans_send;
int ack_seq;
int ack2_send;


/**
 * RTP packet header. Note that all RTP functions here will work with this
 * header in network byte order.
 */
#pragma pack(1)
```

```
struct pjmedia_rtp_hdr
{
#if defined(PJ_IS_BIG_ENDIAN) && (PJ_IS_BIG_ENDIAN!=0)
    pj_uint16_t v:2;          /**< packet type/version          */
    pj_uint16_t p:1;          /**< padding flag           */
    pj_uint16_t x:1;          /**< extension flag               */
    pj_uint16_t cc:4;         /**< CSRC count                   */
    pj_uint16_t m:1;          /**< marker bit                   */
    pj_uint16_t pt:7;         /**< payload type           */
#else
    pj_uint16_t cc:4;         /**< CSRC count                   */
    pj_uint16_t x:1;          /**< header extension flag        */
    pj_uint16_t p:1;          /**< padding flag           */
    pj_uint16_t v:2;          /**< packet type/version          */
    pj_uint16_t pt:7;         /**< payload type           */
    pj_uint16_t m:1;          /**< marker bit                   */
#endif
    pj_uint16_t seq;          /**< sequence number              */
    pj_uint32_t ts;           /**< timestamp                    */
    pj_uint32_t ssrc;         /**< synchronization source       */
};
#pragma pack()

/**
 * @see pjmedia_rtp_hdr
 */
typedef struct pjmedia_rtp_hdr pjmedia_rtp_hdr;


/**
 * RTP extendsion header.
 */
struct pjmedia_rtp_ext_hdr
{
    pj_uint16_t   profile_data;     /**< Profile data.          */
    pj_uint16_t   length;           /**< Length.                */
};

/**
 * @see pjmedia_rtp_ext_hdr
 */
typedef struct pjmedia_rtp_ext_hdr pjmedia_rtp_ext_hdr;


#pragma pack(1)

/**
 * Declaration for DTMF telephony-events (RFC2833).
 */
struct pjmedia_rtp_dtmf_event
{
    pj_uint8_t    event;          /**< Event type ID.     */
    pj_uint8_t    e_vol;          /**< Event volume.      */
    pj_uint16_t   duration;   /**< Event duration.    */
};

/**
```

```
 * @see pjmedia_rtp_dtmf_event
 */
typedef struct pjmedia_rtp_dtmf_event pjmedia_rtp_dtmf_event;

#pragma pack()


/**
 * A generic sequence number management, used by both RTP and RTCP.
 */
struct pjmedia_rtp_seq_session
{
    pj_uint16_t       max_seq;          /**< Highest sequence number heard
*/
    pj_uint32_t       cycles;     /**< Shifted count of seq number cycles */
    pj_uint32_t       base_seq;         /**< Base seq number
*/
    pj_uint32_t       bad_seq;        /**< Last 'bad' seq number + 1
*/
    pj_uint32_t       probation;      /**< Sequ. packets till source is valid
*/
};

/**
 * @see pjmedia_rtp_seq_session
 */
typedef struct pjmedia_rtp_seq_session pjmedia_rtp_seq_session;


/**
 * RTP session descriptor.
 */
struct pjmedia_rtp_session
{
    pjmedia_rtp_hdr         out_hdr;    /**< Saved hdr for outgoing pkts.
*/
    pjmedia_rtp_seq_session seq_ctrl;   /**< Sequence number management.
*/
    pj_uint16_t             out_pt; /**< Default outgoing payload type. */
    pj_uint32_t             out_extseq; /**< Outgoing extended seq #.
*/
    pj_uint32_t             peer_ssrc;  /**< Peer SSRC.
*/
    pj_uint32_t             received;   /**< Number of received packets.
*/
};

/**
 * @see pjmedia_rtp_session
 */
typedef struct pjmedia_rtp_session pjmedia_rtp_session;


/**
 * This structure is used to receive additional information about the
 * state of incoming RTP packet.
 */
```

```
struct pjmedia_rtp_status
{
    union {
      struct flag {
          int     bad:1;          /**< General flag to indicate that sequence
is
                          bad, and application should not process
                          this packet. More information will be given
                          in other flags.                  */
          int badpt:1;    /**< Bad payload type.                      */
          int badssrc:1;  /**< Bad SSRC                           */
          int     dup:1;          /**< Indicates duplicate packet
*/
          int     outorder:1; /**< Indicates out of order packet
*/
          int     probation:1;/**< Indicates that session is in probation
                          until more packets are received.       */
          int     restart:1;  /**< Indicates that sequence number has made
                          a large jump, and internal base sequence
                          number has been adjusted.              */
      } flag;                 /**< Status flags.                          */

      pj_uint16_t value;  /**< Status value, to conveniently address all
                          flags.                                 */

    } status;               /**< Status information union.              */

    pj_uint16_t   diff;     /**< Sequence number difference from previous
                          packet. Normally the value should be 1.
                          Value greater than one may indicate packet
                          loss. If packet with lower sequence is
                          received, the value will be set to zero.
                          If base sequence has been restarted, the
                          value will be one.                     */
};


/**
 * RTP session settings.
 */
typedef struct pjmedia_rtp_session_setting
{
    pj_uint8_t          flags;      /**< Bitmask flags to specify whether such
                                      field is set. Bitmask contents are:
                                    (bit #0 is LSB)
                                    bit #0: default payload type
                                    bit #1: sender SSRC
                                    bit #2: sequence
                                    bit #3: timestamp                */
    int                 default_pt;    /**< Default payload type.
*/
    pj_uint32_t         sender_ssrc;   /**< Sender SSRC.
*/
    pj_uint16_t         seq;        /**< Sequence.                      */
    pj_uint32_t         ts;         /**< Timestamp.                     */
} pjmedia_rtp_session_setting;
```

```
/**
 * @see pjmedia_rtp_status
 */
typedef struct pjmedia_rtp_status pjmedia_rtp_status;


/**
 * This function will initialize the RTP session according to given
parameters.
 *
 * @param ses           The session.
 * @param default_pt    Default payload type.
 * @param sender_ssrc   SSRC used for outgoing packets, in host byte order.
 *
 * @return        PJ_SUCCESS if successfull.
 */
PJ_DECL(pj_status_t) pjmedia_rtp_session_init( pjmedia_rtp_session *ses,
                                       int default_pt,
                                       pj_uint32_t sender_ssrc );

/**
 * This function will initialize the RTP session according to given
parameters
 * defined in RTP session settings.
 *
 * @param ses           The session.
 * @param settings      RTP session settings.
 *
 * @return        PJ_SUCCESS if successfull.
 */
PJ_DECL(pj_status_t) pjmedia_rtp_session_init2(
                           pjmedia_rtp_session *ses,
                           pjmedia_rtp_session_setting settings);


/**
 * Create the RTP header based on arguments and current state of the RTP
 * session.
 *
 * @param ses           The session.
 * @param pt            Payload type.
 * @param m       Marker flag.
 * @param payload_len   Payload length in bytes.
 * @param ts_len  Timestamp length.
 * @param rtphdr  Upon return will point to RTP packet header.
 * @param hdrlen  Upon return will indicate the size of RTP packet header
 *
 * @return        PJ_SUCCESS if successfull.
 */
PJ_DECL(pj_status_t) pjmedia_rtp_encode_rtp( pjmedia_rtp_session *ses,
                                     int pt, int m,
                                     int payload_len, int ts_len,
                                     const void **rtphdr,
                                     int *hdrlen );

/**
```

```
 * This function decodes incoming packet into RTP header and payload.
 * The decode function is guaranteed to point the payload to the correct
 * position regardless of any options present in the RTP packet.
 *
 * Note that this function does not modify the returned RTP header to
 * host byte order.
 *
 * @param ses           The session.
 * @param pkt           The received RTP packet.
 * @param pkt_len The length of the packet.
 * @param hdr           Upon return will point to the location of the RTP
 *                header inside the packet. Note that the RTP header
 *                will be given back as is, meaning that the fields
 *                will be in network byte order.
 * @param payload Upon return will point to the location of the
 *                payload inside the packet.
 * @param payloadlen    Upon return will indicate the size of the payload.
 *
 * @return        PJ_SUCCESS if successfull.
 */
PJ_DECL(pj_status_t) pjmedia_rtp_decode_rtp( pjmedia_rtp_session *ses,
                                     const void *pkt, int pkt_len,
                                     pjmedia_rtp_hdr **hdr,
                                     const void **payload,
                                     unsigned *payloadlen);

/**
 * Call this function everytime an RTP packet is received to check whether
 * the packet can be received and to let the RTP session performs its
internal
 * calculations.
 *
 * @param ses         The session.
 * @param hdr         The RTP header of the incoming packet. The header must
 *              be given with fields in network byte order.
 * @param seq_st    Optional structure to receive the status of the RTP
packet
 *              processing.
 */
PJ_DECL(void) pjmedia_rtp_session_update( pjmedia_rtp_session *ses,
                                  const pjmedia_rtp_hdr *hdr,
                                  pjmedia_rtp_status *seq_st);


/**
 * Call this function everytime an RTP packet is received to check whether
 * the packet can be received and to let the RTP session performs its
internal
 * calculations.
 *
 * @param ses         The session.
 * @param hdr         The RTP header of the incoming packet. The header must
 *              be given with fields in network byte order.
 * @param seq_st    Optional structure to receive the status of the RTP
packet
 *              processing.
```

```
 * @param check_pt  Flag to indicate whether payload type needs to be
validate.
 *
 * @see pjmedia_rtp_session_update()
 */
PJ_DECL(void) pjmedia_rtp_session_update2(pjmedia_rtp_session *ses,
                                const pjmedia_rtp_hdr *hdr,
                                pjmedia_rtp_status *seq_st,
                                pj_bool_t check_pt);


/*
 * INTERNAL:
 */

/**
 * Internal function for creating sequence number control, shared by RTCP
 * implementation.
 *
 * @param seq_ctrl  The sequence control instance.
 * @param seq         Sequence number to initialize.
 */
void pjmedia_rtp_seq_init(pjmedia_rtp_seq_session *seq_ctrl,
                  pj_uint16_t seq);


/**
 * Internal function update sequence control, shared by RTCP implementation.
 *
 * @param seq_ctrl      The sequence control instance.
 * @param seq           Sequence number to update.
 * @param seq_status    Optional structure to receive additional information
 *                about the packet.
 */
void pjmedia_rtp_seq_update( pjmedia_rtp_seq_session *seq_ctrl,
                      pj_uint16_t seq,
                      pjmedia_rtp_status *seq_status);

/**
 * @}
 */

PJ_END_DECL


#endif      /* __PJMEDIA_RTP_H__ */
```

Appendix 9: rtp.c source code (raw)

```
/* $Id: rtp.c 2394 2008-12-23 17:27:53Z bennylp $ */
/*
 * Copyright (C) 2008-2009 Teluu Inc. (http://www.teluu.com)
 * Copyright (C) 2003-2008 Benny Prijono <benny@prijono.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
#include <pjmedia/rtp.h>
#include <pjmedia/errno.h>
#include <pj/log.h>
#include <pj/sock.h>    /* pj_htonx, pj_htonx */
#include <pj/assert.h>
#include <pj/rand.h>
#include <pj/string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define THIS_FILE   "rtp.c"

#define RTP_VERSION     2

#define RTP_SEQ_MOD     (1 << 16)
#define MAX_DROPOUT     ((pj_int16_t)3000)
#define MAX_MISORDER    ((pj_int16_t)100)
#define MIN_SEQUENTIAL  ((pj_int16_t)2)
#define IN_FILE                 "input.txt"
#define OUT_FILE        "output.txt"
#define DELAY_TIME              3
#define SEND_DELAY              1
FILE *infp;       //input file
FILE *outfp;      //output file
time_t start_time;
time_t cur_time;
time_t call_start;


static void pjmedia_rtp_seq_restart(pjmedia_rtp_seq_session *seq_ctrl,
                                    pj_uint16_t seq);
```

```
PJ_DEF(pj_status_t) pjmedia_rtp_session_init( pjmedia_rtp_session *ses,
                                      int default_pt,
                                      pj_uint32_t sender_ssrc )
{
    PJ_LOG(5, (THIS_FILE,
              "pjmedia_rtp_session_init: ses=%p, default_pt=%d, ssrc=0x%x",
              ses, default_pt, sender_ssrc));

    /* Check RTP header packing. */
    if (sizeof(struct pjmedia_rtp_hdr) != 12) {
      pj_assert(!"Wrong RTP header packing!");
      return PJMEDIA_RTP_EINPACK;
    }

    /* If sender_ssrc is not specified, create from random value. */
    if (sender_ssrc == 0 || sender_ssrc == (pj_uint32_t)-1) {
      sender_ssrc = pj_htonl(pj_rand());
    } else {
      sender_ssrc = pj_htonl(sender_ssrc);
    }

    /* Initialize session. */
    pj_bzero(ses, sizeof(*ses));

    /* Initial sequence number SHOULD be random, according to RFC 3550. */
    /* According to RFC 3711, it should be random within 2^15 bit */
    ses->out_extseq = pj_rand() & 0x7FFF;
    ses->peer_ssrc = 0;

    /* Build default header for outgoing RTP packet. */
    ses->out_hdr.v = RTP_VERSION;
    ses->out_hdr.p = 0;
    ses->out_hdr.x = 0;
    ses->out_hdr.cc = 0;
    ses->out_hdr.m = 0;
    ses->out_hdr.pt = (pj_uint8_t) default_pt;
    ses->out_hdr.seq = (pj_uint16_t) pj_htons( (pj_uint16_t)ses->out_extseq
);
    ses->out_hdr.ts = 0;
    ses->out_hdr.ssrc = sender_ssrc;

    /* Keep some arguments as session defaults. */
    ses->out_pt = (pj_uint16_t) default_pt;

      /* open files to read write */
      infp = fopen(IN_FILE, "r");
      outfp = fopen(OUT_FILE, "w+");

      //initalize covert values
      ack_rcvd=1;
      ts_mod_last=0;
      ack_send=0;
      ready_send=0;
      time(&call_start);
      retrans_rcv = 0;
      retrans_send = 0;
```

```
    return PJ_SUCCESS;
}

PJ_DEF(pj_status_t) pjmedia_rtp_session_init2(
                           pjmedia_rtp_session *ses,
                           pjmedia_rtp_session_setting settings)
{
    pj_status_t status;
    int             pt = 0;
    pj_uint32_t     sender_ssrc = 0;

    if (settings.flags & 1)
      pt = settings.default_pt;
    if (settings.flags & 2)
      sender_ssrc = settings.sender_ssrc;

    status = pjmedia_rtp_session_init(ses, pt, sender_ssrc);
    if (status != PJ_SUCCESS)
      return status;

    if (settings.flags & 4) {
      ses->out_extseq = settings.seq;
      ses->out_hdr.seq = pj_htons((pj_uint16_t)ses->out_extseq);
    }
    if (settings.flags & 8)
      ses->out_hdr.ts = pj_htonl(settings.ts);

    return PJ_SUCCESS;
}


PJ_DEF(pj_status_t) pjmedia_rtp_encode_rtp( pjmedia_rtp_session *ses,
                                    int pt, int m,
                                    int payload_len, int ts_len,
                                    const void **rtphdr, int *hdrlen )
{
    PJ_UNUSED_ARG(payload_len);

      //set prior ts modification value
      if(ts_mod_old<160 && ts_mod_old>=(-1)){
      }
      else{
            ts_mod_old=0;
      }

      //set ts_mod to zero in case no new data
      ts_mod=0;

      time(&cur_time);

      //delay mechanism if needed at start.  disabled if readysend = (1)
      if(ready_send==0){
            if(difftime(cur_time, call_start)>SEND_DELAY){
                  ready_send=1;
            }
      }
```

```
      //transmit new data if last recieved
      else if(ack_rcvd == 1){
            //set timestamp modification for current packet
            ts_mod=(getc(infp));

            //reset null to 128
            if (ts_mod == 0) {
                  ts_mod=128;
            }

            //adjust if EOF. no ack for
            if (ts_mod == -1){
                  ts_mod=0;
            }
            else{
                  //set transmitted value to last sent
                  ts_mod_last = ts_mod;
            }
      }


    /* Update timestamp */
    ses->out_hdr.ts = pj_htonl(pj_ntohl(ses->out_hdr.ts)+ts_len+ts_mod-
ts_mod_old);

      //undo timestamp offset
      ts_mod_old=ts_mod;



    /* If payload_len is zero, bail out.
     * This is a clock frame; we're not really transmitting anything.
     */
    //if (payload_len == 0)
      //return PJ_SUCCESS;

    /* Update session. */
    ses->out_extseq++;

    /* Create outgoing header. */
    ses->out_hdr.pt = (pj_uint8_t) ((pt == -1) ? ses->out_pt : pt);
    ses->out_hdr.m = (pj_uint16_t) m;
    ses->out_hdr.seq = pj_htons( (pj_uint16_t) ses->out_extseq);

    /* Return values */
    *rtphdr = &ses->out_hdr;
    *hdrlen = sizeof(pjmedia_rtp_hdr);

    return PJ_SUCCESS;
}


PJ_DEF(pj_status_t) pjmedia_rtp_decode_rtp( pjmedia_rtp_session *ses,
                                    const void *pkt, int pkt_len,
                                    pjmedia_rtp_hdr **hdr,
                                    const void **payload,
                                    unsigned *payloadlen)
```

```
{
    int offset;
      pj_uint32_t ts_diff;
      pj_uint32_t ts_tmp;

    PJ_UNUSED_ARG(ses);

    /* Assume RTP header at the start of packet. We'll verify this later. */
    *hdr = (pjmedia_rtp_hdr*)pkt;

    /* Check RTP header sanity. */
    if ((*hdr)->v != RTP_VERSION) {
      return PJMEDIA_RTP_EINVER;
    }

      /*decode covert data*/
      ts_tmp = pj_ntohl((*hdr)->ts);
      ts_diff=( ts_tmp % 160);
      else if ((ts_diff < 130) && (ts_diff > 0)){
            if(ts_diff == 128){
                  ts_diff=0;
            }
            putchar(ts_diff);
            fputc(ts_diff,outfp);
            ack_send=1;
      }
      //adjust timestamp to expected value for program to process
      (*hdr)->ts = pj_htonl(ts_tmp-ts_diff);

    /* Payload is located right after header plus CSRC */
    offset = sizeof(pjmedia_rtp_hdr) + ((*hdr)->cc * sizeof(pj_uint32_t));

    /* Adjust offset if RTP extension is used. */
    if ((*hdr)->x) {
      pjmedia_rtp_ext_hdr *ext = (pjmedia_rtp_ext_hdr*)
                                 (((pj_uint8_t*)pkt) + offset);
      offset += (pj_ntohs(ext->length) * sizeof(pj_uint32_t));
    }

    /* Check that offset is less than packet size */
    if (offset > pkt_len)
      return PJMEDIA_RTP_EINLEN;

    /* Find and set payload. */
    *payload = ((pj_uint8_t*)pkt) + offset;
    *payloadlen = pkt_len - offset;

    return PJ_SUCCESS;
}


PJ_DEF(void) pjmedia_rtp_session_update( pjmedia_rtp_session *ses,
                                         const pjmedia_rtp_hdr *hdr,
                                         pjmedia_rtp_status *p_seq_st)
{
    pjmedia_rtp_session_update2(ses, hdr, p_seq_st, PJ_TRUE);
}
```

```
PJ_DEF(void) pjmedia_rtp_session_update2( pjmedia_rtp_session *ses,
                                  const pjmedia_rtp_hdr *hdr,
                                  pjmedia_rtp_status *p_seq_st,
                                  pj_bool_t check_pt)
{
    pjmedia_rtp_status seq_st;

    /* for now check_pt MUST be either PJ_TRUE or PJ_FALSE.
     * In the future we might change check_pt from boolean to
     * unsigned integer to accommodate more flags.
     */
    pj_assert(check_pt==PJ_TRUE || check_pt==PJ_FALSE);

    /* Init status */
    seq_st.status.value = 0;
    seq_st.diff = 0;

    /* Check SSRC. */
    if (ses->peer_ssrc == 0) ses->peer_ssrc = pj_ntohl(hdr->ssrc);

    if (pj_ntohl(hdr->ssrc) != ses->peer_ssrc) {
      seq_st.status.flag.badssrc = 1;
      ses->peer_ssrc = pj_ntohl(hdr->ssrc);
    }

    /* Check payload type. */
    if (check_pt && hdr->pt != ses->out_pt) {
      if (p_seq_st) {
          p_seq_st->status.value = seq_st.status.value;
          p_seq_st->status.flag.bad = 1;
          p_seq_st->status.flag.badpt = 1;
      }
      return;
    }

    /* Initialize sequence number on first packet received. */
    if (ses->received == 0)
      pjmedia_rtp_seq_init( &ses->seq_ctrl, pj_ntohs(hdr->seq) );

    /* Check sequence number to see if remote session has been restarted. */
    pjmedia_rtp_seq_update( &ses->seq_ctrl, pj_ntohs(hdr->seq), &seq_st);
    if (seq_st.status.flag.restart) {
      ++ses->received;

    } else if (!seq_st.status.flag.bad) {
      ++ses->received;
    }

    if (p_seq_st) {
      p_seq_st->status.value = seq_st.status.value;
      p_seq_st->diff = seq_st.diff;
    }
}
```

```
void pjmedia_rtp_seq_restart(pjmedia_rtp_seq_session *sess, pj_uint16_t seq)
{
    sess->base_seq = seq;
    sess->max_seq = seq;
    sess->bad_seq = RTP_SEQ_MOD + 1;
    sess->cycles = 0;
}


void pjmedia_rtp_seq_init(pjmedia_rtp_seq_session *sess, pj_uint16_t seq)
{
    pjmedia_rtp_seq_restart(sess, seq);

    sess->max_seq = (pj_uint16_t) (seq - 1);
    sess->probation = MIN_SEQUENTIAL;
}


void pjmedia_rtp_seq_update( pjmedia_rtp_seq_session *sess,
                        pj_uint16_t seq,
                        pjmedia_rtp_status *seq_status)
{
    pj_uint16_t udelta = (pj_uint16_t) (seq - sess->max_seq);
    pjmedia_rtp_status st;

    /* Init status */
    st.status.value = 0;
    st.diff = 0;

    /*
     * Source is not valid until MIN_SEQUENTIAL packets with
     * sequential sequence numbers have been received.
     */
    if (sess->probation) {

      st.status.flag.probation = 1;

        if (seq == sess->max_seq+ 1) {
          /* packet is in sequence */
          st.diff = 1;
          sess->probation--;
            sess->max_seq = seq;
            if (sess->probation == 0) {
            st.status.flag.probation = 0;
            }
      } else {

          st.diff = 0;

          st.status.flag.bad = 1;
          if (seq == sess->max_seq)
            st.status.flag.dup = 1;
          else
            st.status.flag.outorder = 1;

          sess->probation = MIN_SEQUENTIAL - 1;
          sess->max_seq = seq;
```

```
        }


    } else if (udelta == 0) {

      st.status.flag.dup = 1;

    } else if (udelta < MAX_DROPOUT) {
      /* in order, with permissible gap */
      if (seq < sess->max_seq) {
          /* Sequence number wrapped - count another 64K cycle. */
          sess->cycles += RTP_SEQ_MOD;
        }
        sess->max_seq = seq;

      st.diff = udelta;

    } else if (udelta <= (RTP_SEQ_MOD - MAX_MISORDER)) {
      /* the sequence number made a very large jump */
        if (seq == sess->bad_seq) {
          /*
           * Two sequential packets -- assume that the other side
           * restarted without telling us so just re-sync
           * (i.e., pretend this was the first packet).
           */
          pjmedia_rtp_seq_restart(sess, seq);
          st.status.flag.restart = 1;
          st.status.flag.probation = 1;
          st.diff = 1;
      }
        else {
          sess->bad_seq = (seq + 1) & (RTP_SEQ_MOD-1);
            st.status.flag.bad = 1;
          st.status.flag.outorder = 1;
        }
    } else {
      /* old duplicate or reordered packet.
       * Not necessarily bad packet (?)
       */
      st.status.flag.outorder = 1;
    }


    if (seq_status) {
      seq_status->diff = st.diff;
      seq_status->status.value = st.status.value;
    }
}
```