Real-Time Voice Transmission Over the Internet

Laboratory Assignment 6

ECE 371, Section BW

Srikanth Kandula (kandula@uiuc.edu)
Sankalp Singh (ssingh7@uiuc.edu)

November 24, 2001

1 Introduction

The goal of this experiment was to quantify the performance of real-time audio transmissions in Internet for different transmission protocols, interpolation methods and degrees of interleaving. For real-time audio over TCP, we also attempt to measure the efficacy of using delay jitter buffers to smoothen out the application-perceived jitter. In each of the following sections, we discuss in detail the individual steps involved in this laboratory exercise.

2 Experiences with codecs

Code implementing the APCM compression mechanism had been provided as a start. However, these had to be slightly modified to run on our local machine. Specifically, we had to remove some ioctl calls that were attempting to access DUPLEX functionality on the speakers and other similar things. We feel that this was either due to lack of proper support in the speakers or due to incorrect configuration of the host kernel. Since the functionality implemented by these lines of code were not necessary for the assignment, they were removed without further ado.

3 UDP Interleaving, Inerpolation and Re-constructed quality estimation

Two applications were built for this experiment. The "**Sender**" was a multi-threaded application that was built upon the *sender.cc* provided with the initial code. It consisted of two threads described below.

- **Sender Thread**: This thread reads the multimedia input from the microphone, performs the desired interleaving, encodes each stream using the code given, encapsulates the encoded streams and sends them to the remote UDP echo port. The interleaving mechanism is described in more detail in ∮3.1. Since we needed to run the entire experiment over a period of 24 hours and we wished to evaluate the efficacy of different interpolation techniques, we *a priori* recorded two different voice samples into as many files. The sender thread read from one of these files instead of directly reading from the files. In case it reaches the EOF, it simply starts from the beginning of the file again. The difference in the voice samples and the reasons for the choice are described in ∮3.2 in more detail. We experimentally determined the rate at which data is recorded off the microphone to be about 1 kB per 35 msecs. To ensure that sender thread sends data at the same rate as it would have in case the data was being read from the microphone directly, it is made to sleep for about 15 msec for 512 B of data read from the file.

- **Receiving and Forwarding Thread**: This thread simply read the data echoed back from the echo server and forwarded it to the Receiver application. Since UDP is unreliable, we needed to characterize a packet loss. For this purpose, before sending the encoded data, sender thread also encapsulates it in a header containing a sequence number. The receiving thread also maintains the next sequence number it ought to recieve (which is one more than the last packet received). In case the sequence number of a packet received is more than the expected number, all the interveaning packets are assumed lost and the expected sequence number is increased to one more than the sequence number of the current packet. In case the current sequence number is less than the expected sequence number, it is simply ignored. Otherwise the current packet is forwarded to the receiver application for playback. Hence the receiver application would always receive packets with increasing sequence numbers and can characterize missing sequence numbers as missing.

The "**Receiver**" was a single-threaded application that received the UDP packets forwarded by the forwarding thread of the sender. It decoded the received packets, and in case of losses, performed the relevant interpolation (linear or quadratic), reconstructed the voice stream and played it back. It also wrote the final reconstructed voice stream to file for use in SNR calculations. $\oint$3.2 describes in more detail how interpolation was performed.

3.1 Interleaving the recorded voice stream

Interleaving was performed to safeguard against the losses in the network and allow for stream reconstruction. We used the interleaving degree of 2 for the domestic sites and degree of 4 for international sites.

To perform the desired degree of interleaving (which was defined as a macro in *common.h*), we decomposed the buffer containing the recorded sample into 2 (or 4) buffers with each alternate byte (or each byte in sets of 4 bytes) going into different interleaved buffers. This was possible because each sample (one byte of recorded data) was independent of the others. After this, we encoded each of the smaller buffers using (a slightly modified to adjust the new buffer sizes) *encode* routine. Each of these smaller encoded data are then encapsulated with sequence numbers and sent to the echo server. When the receiver application has reconstructed the stream, the packets are re-integrated together (using algorithm just reverse of one used for interleaving) and played.

One point to note is when was interpolation performed and when was silence fed into the stream being played. We fed silence into the stream being played only if all the interleaved packets derived from the same intial recorded buffer are lost. Hence, in case of 2 way interleaving, this would be when both packets of an interlaved pair are lost and in case of 4 way interleaving, this would be the case when all four packets in the interleaved quartet are lost. If fewer packets are lost, we attempt to recontruct the missing ones using those received. This is the reason for choosing 4 way interleaving in scenarios with more loss—at least one of the streams would be received and some reconstruction would be performed.

The relevant source code is shown in ∮B.1 and ∮B.2.

3.2 Interpolation for stream reconstruction

We tested two different interpolation schemes to perform reconstruction of missing samples: linear interpolation and quadratic interpolation.

- **Linear interpolation**: When using linear interpolation, any missing sample (byte) was generated using the average of the nearest samples before and after the missing sample. As described above, this reconstruction was performed only if at least one sample in the interleaved tuple was received. Hence, if S_l is the nearest non-missing sample before the missing sample in the un-interleaved voice stream, and S_r is nearest non-missing sample after the missing sample in the un-interleaved stream, the missing sample S was calculated as:

  $$S = \frac{S_l+S_r}{2}$$

  The code is shown in ∮B.4.1. It is a generic, and rather tricky code which works for any degree of interpolation without writing out separate cases for handling all the permutaitons of sample misses.

- **Quadratic Interpolation**: When using quadratic interpolation, any missing sample (byte) was generated using three nearby non-missing samples. Again, the reconstruction was performed only if at least one sample in the interleaved tuple was received. If S_l is the nearest non-missing sample before the missing sample in the un-interleaved voice stream, and S_{r1} and S_{r2} are the nearest and next-nearest samples after the missing sample in the un-interleaved stream, then the missing sample S is generated using the standard quadratic interpolation formula:

$$S = S_l + \frac{1}{2}(S_{r1} - S_l) - \frac{1}{8}((S_{r2} - S_{r1}) - (S_{r1} - S_l))$$

The code is shown in ∮B.4.2. Again, it is a generic (and tricky) code which works for any degree of interpolation without handling each possible combination of misses separately.

Using two kinds of voice samples: We observed early during our experiments that linear interpolation worked better for voice streams which had a lot of discontinuity and more of a piecewise linear kind of signal. On the other hand, quadratic sampling which looked at three nearby samples for reconstruction, worked better for voice streams which had a smoother kind of sound with signal changing as a smoother (at least differentiable) funtion. To verify our intial observations, we performed our experiments on two streams we had recorded into files: discontinuous stream, which was recorded while counting from 1 to 100 with a pause between two numbers, and a continous stream which consists of a constant humming sound with smoother variations. As expected, linear interpolation works better for the former and quadratic works better for the latter. These observations are further described in the section on graphs and explainations, ∮7.3.

3.3 SNR Estimation

Signal to noise ratio was used as an estimate of the quality of the reconstructed signal. It was calculated as:

$$SNR = 10log_{10}\frac{\sum s^2}{\sum(s-\hat{s})^2}$$

where s is the original signal and $\hat{s}$ is the received reconstructed signal. We use original signal values from the original recorded sound file and reconstructed values written and played by the receiver application. The SNR values are calculated for each hourly sample.

SNR is a measure of efficacy of the interpolation scheme and is also greatly influenced by the amount of losses, esp. the ones that could not be reconstructed.

4 TCP Delay jitter estimation

Two applications were built for this experiment. The "**Sender**" was a multi-threaded application that spawns three threads which are described below.

- **Sender Thread**: This thread reads multimedia input from the microphone, encodes it using the code given, encapsulates the encoded stream and sends the encapsulated packet stream to a remote TCP Echo port. We experimentally obtained the rate at which data is recorded off the microphone to be about 1kB of data in 35msec. Thus this thread safely sleeps for about 10mSec after sending out each 512B packet(i.e. 256B after encoding). We used pthread_cond_timedwait to implement sleep in threads. This aided the functionality of the multi-threaded application as sleep in a thread guarantees that time quanta would be available for other threads. We experimentally verified that no audio samples were being lost due to this sleep phase. Though this phase was not necessary for checking the correctness of this application, this was necessary during the later 24hr data collection phase in order to simulate real-time audio capture by reading original data from a file.

- **Receiver Thread**: The main impediment to the use of TCP for real-time multimedia communication is delay jitter. We built application level jitter buffers in order to smoothen this jitter. Thus reading/writing into the buffer is a traditional producer-consumer problem and both these actions should be independent of each other provided certain criterion were satisfied. This is explained in detail later. This thread was responsible for reading the data echoed by the remote TCP Echo port and storing it into the buffer (Producer). Blocking reads were used.

- **Forwarder Thread**: This thread is the consumer of the buffer mentioned above. Data packets queued by the receiver thread are read out periodically and forwarded to the application. The periodicity of this thread was of the same order as the periodicity of the Sender thread and was meant to simulate the reads of a multimedia application.

The "**Receiver**" is a single-thread that simulates a multimedia application. It was responsible for reading the stream data sent out by the forwarding thread and playing the stream. Additionally jitter of this stream was computed to measure the efficiency of our jitter buffer implementation. Jitter measured here will be referred in later discussion as "Application-perceived jitter".

4.1 Implementation of the Jitter Buffer

The jitter buffer was implemented with the following semantics for a given jitter buffer size.

- If buffer is empty, wait until atleast “jitter buffer size” number of packets are available before forwarding to the application.
- If buffer is not empty and we are not in a wait phase, forward packets to the application periodically.
- If we are in wait phase and the number of packets in the buffer exceeds “jitter buffer size” start forwarding.

Thus the forwarder thread could be in two distinct phases, “forwarding” and “waiting”. Initially it starts out in the waiting phase, simulated by pthread_cond_wait. When the receiver thread inserts packets into the buffer, it checks whether the buffer size is greater than the jitter buffer size estimated and wakes the forwarder thread (if necessary). The forwarder thread keeps forwarding until the buffer is empty, at which time it goes into the waiting phase. Thus the rationale behind this was that optimally the jitter buffer is never empty, so the end application would not perceive any jitter. On the other hand, in the eventuality that the jitter buffer is empty, the application is forced to wait for atleast jitter buffer size number of packets to queue up, such that only one packet is excessively delayed (and high jitter) . However all subsequent packets would experience no jitter.

4.2 Estimation of Jitter

Jitter was estimated as the second order difference between the delays. The average inter-packet delay was maintained. The difference between the inter-packet delay seen after the current packet and the average inter-packet delay is the current jitter. Thus jitter is the second order difference between the inter-packet arrival times. We studied the algorithm for jitter buffer size estimation presented in Liang and Cheriton’s paper but concluded that a newer scheme based on average jitter estimates would be better. In the next section, we present both these schemes and argue for our scheme.

4.3 Calculation of jitter buffer size

Liang and Cheriton [1] compute jitter buffer size as $((\frac{3}{2}*RTT+3*period)+delta)*rate$ number of packets or bytes . The rationale behind this is that the worst-case jitter between two packets, the first of which arrives in time but the second is lost and needs to be re-transmitted using TCP’s fast re-transmit. Effectively, since period is $\frac{1}{rate}$ the above expression reduces to $(\frac{3}{2} * RTT * rate + 3 + delta * rate)$. Thus the above estimate for jitter buffer size is a worst case estimate and suffers from two major problems. On networks which have significantly low error rates, this is an overkill, i.e. the jitter buffer size is larger than necessary. Further on severely congested networks, it could happen that TCP errors could only be recovered through a Retrasmission timeout. The jitter that packets would experience in this case would be greater than the above worst case estimate. Hence, we use an online estimation of the average jitter seen by packets in this connection till now and compute the jitter buffer size as $max(alpha * averagejitter * rate, delta)$. Alpha is a small constant such as 4 and delta is a small constant such as 3. Thus it could be seen that our expression is very close to the above under the condition that average jitter $\leq \frac{3}{2} * RTT$. Further, this is a significantly better estimate as this would be more customized to the current connections state, both better conditions when average jitter is less (very few TCP errors) and worse conditions when average jitter is worse (retransmission time-outs).

The results for this phase of experiments have been shown in ∮7.1. An explanation of each of the result follows each graph.

5 TCP encapsulated in UDP Delay jitter estimation

In this section, we used the code developed in Lab Assignment 5. Code described in detail for the above section could be easily modified for this experiment. The following modifications were necessitated.

- To conform to the semantics of Lab 5, i.e. TCP packets sent from a local sender to a local receiver would be encapsulated in UDP and echoed off a remote UDP Echo port, the sender thread was modified to send out audio packets to a local receiver after setting the correct TCP socket options. Thus the "Sender" is a single-thread, i.e. only the sender thread discussed above.
- A new "Local Receiver" was built consisting of the "Forwarder thread" and the "Receiver thread". Data was received on a local socket, stored in a buffer to smooth jitter and periodically forwarded to a receiver as before.
- No changes were needed for the "Receiver".

The results for this phase of experiments have been shown in ∮7.2. An explanation of each of the result follows each graph.

6 Experiment setup

24 samples were taken, one per hour over a 24hr period. Each sample was obtained by transmitting 5000 packets. The Senders was modified to read from a file rather than reading the live audio. The Receivers were modified to write the received stream of packets into a local file rather than playing it out in real time. The following data was collected for each of the above experiments.

- TCP (Remote TCP Echo and TCP encapsulated in UDP): The "Sender" stored the delay, jitter, number of packets in the buffer and the estimate of the jitter buffer size for each packet received by the Receiver thread. The "Receiver" stored the delay and jitter for each packet received. Packets were not tagged as TCP Streams guarantee in-order-delivery. Thus the two pairs of data were matched based on the Packets id in the sequence recievec. Average jitter, delay and jitter buffer size were computed. These values have been drawn in graphs to illustrate several key aspects.
- UDP: The "Sender" was again modified to read off a locally recorded file rather than from the microphone. The experiments were run for two different files, one of which contained a discontinuous stream (counting of numbers) and the other contained a continuous stream (a single tune). The receiver computed the total number of packets lost among the 5000 sent and computed the SNR's by comparing the original stream with the streams reconstructed using both the interpolation methods.

Data packets were sent out to three international sites using 4 way interleaving for UDP packets and to two domestic sites using 2 way interleaving again for UDP packets. The next section shows the graphs and the comments on the results. The code for this lab has been attached in the Appendix.

7 Graphical results and Comments

7.1 TCP Echo, Jitter Estimations

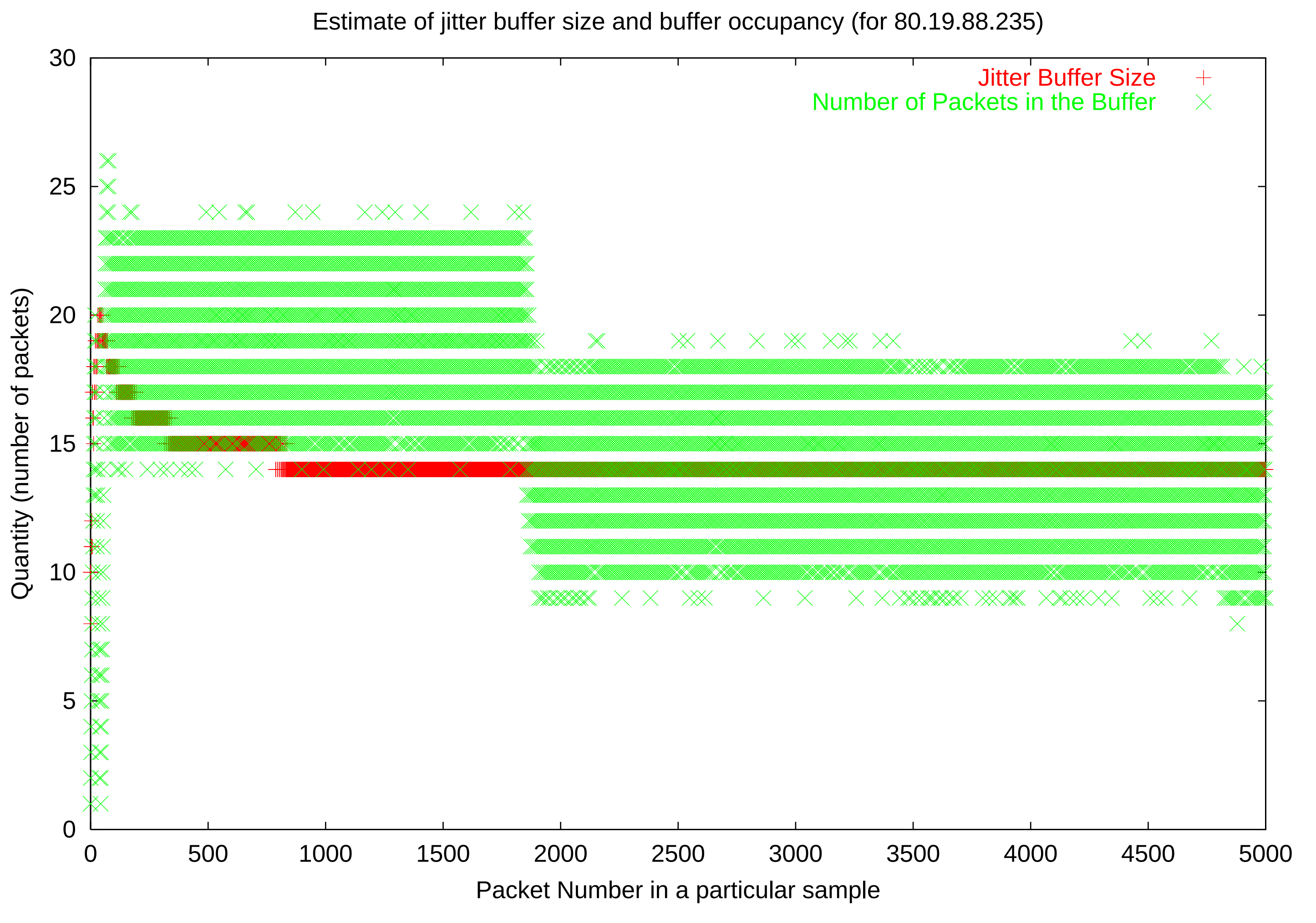
For a given host and a given sample, we plot the jitter from the network and the application perceived jitter for each packet in **Graph 1** above. This graph clearly shows that a large fraction of jiters after smoothing are smaller than the jitter from the network. The jitter as seen from the network also has a significantly larger variance (burstiness) than after smoothing. Thus on the whole not only the average jitter is smaller but a large percentage of the jitters are small. This is for TCP Echo off an international server.

Graph 2

For a given host and a given sample, this plot shows the inter-packet delay and the jitter experienced by packets from the network. The idea is that a significant fraction of packets experience small delays and small jitters. However a small percentage experience large delays and correspondingly larger jitters. This is precisely the problem with TCP and has been corrected using a smoothing buffer as shown in the previous graph.

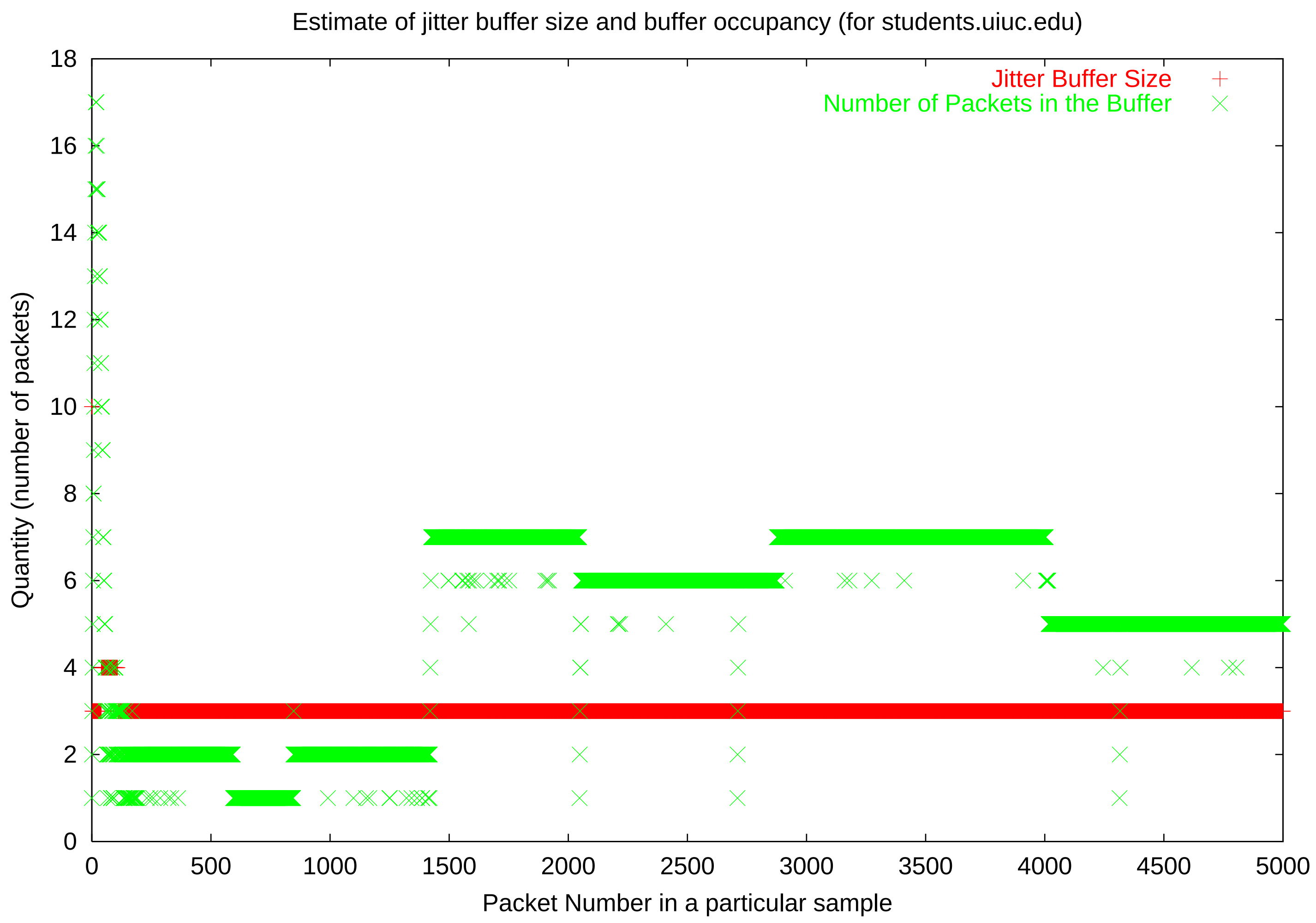
Graph 3

For a given host and a given sample, this plot compares the inter-packet delay between packets received off the network and the inter-packet delay between packets as received by the application. Clearly smoothing causes a slight increase in the inter-packet delay. The point to be noted however is that this inter-packet delay (after smoothing) is conformant with the period of the multimedia application and has been enforced by having the forwarding thread sleep. The usefulness of smoothing is that most of the delays are clustered around the desirable delay which gives credence to the fact that the jitters are very small. Thus smoothing is indeed useful.



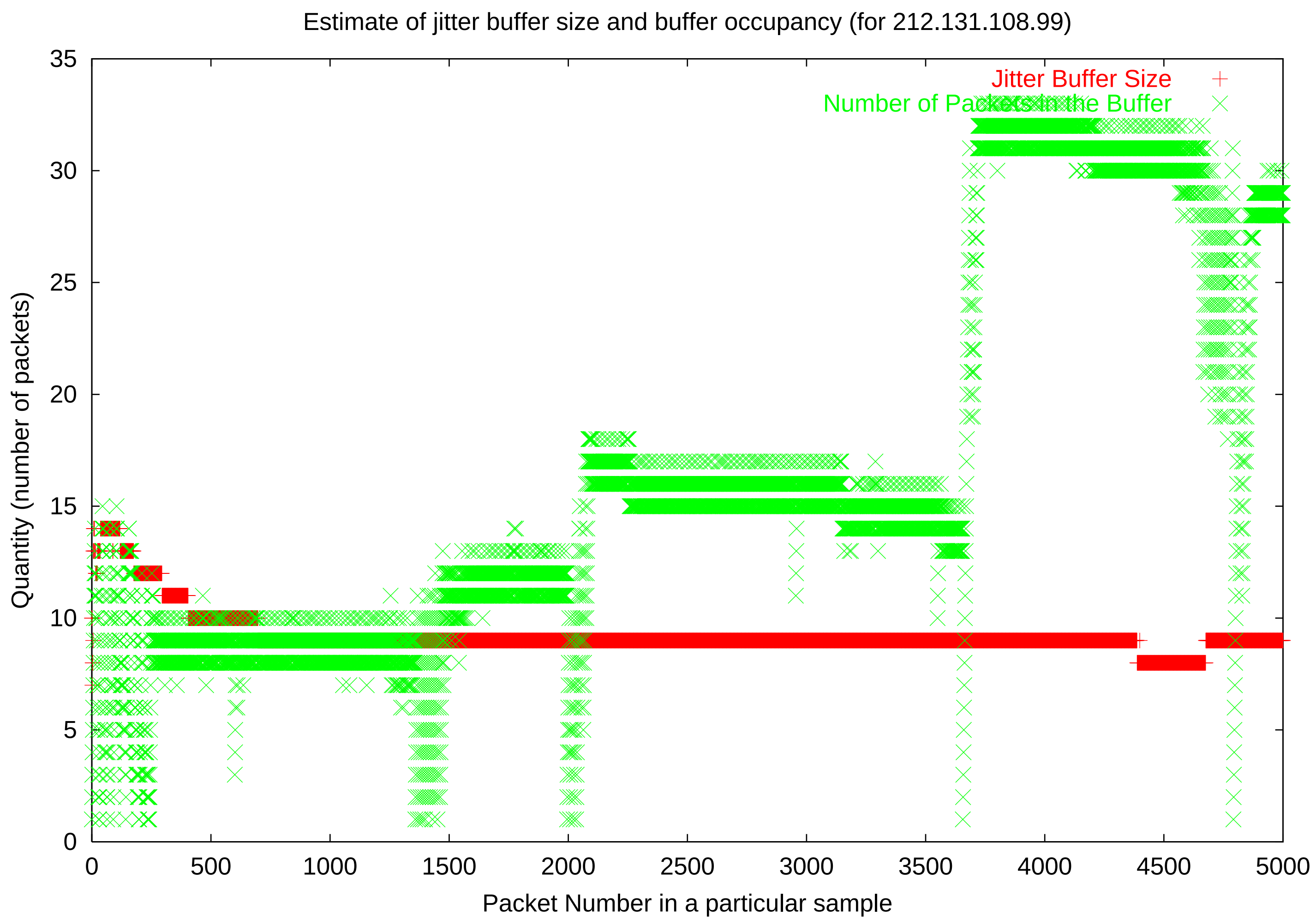
Graph 4

Crucial points to note when enforcing a jitter buffer at the application level, are that the jitter buffer size needs to be small and that the buffer occupancy should not be too high. This graph shows the varying nature of jitter buffer size estimates (based on average jitter seen off the network) and the instantaneous buffer sizes, i.e. the number of packets in the buffer at the arrival of each packet. It could be clearly seen that the estimate of jitter buffer size is about 14 (a small value) and the buffer occupancy throughout the experiment are stable. Instances when the number of packets in the buffer fall to 0 are the instances when the application would perceive larger jitters. The number of such instances is very few, thus it could be seen that smoothing is not very efficient in this case. Compare these results with those in Graph 6.



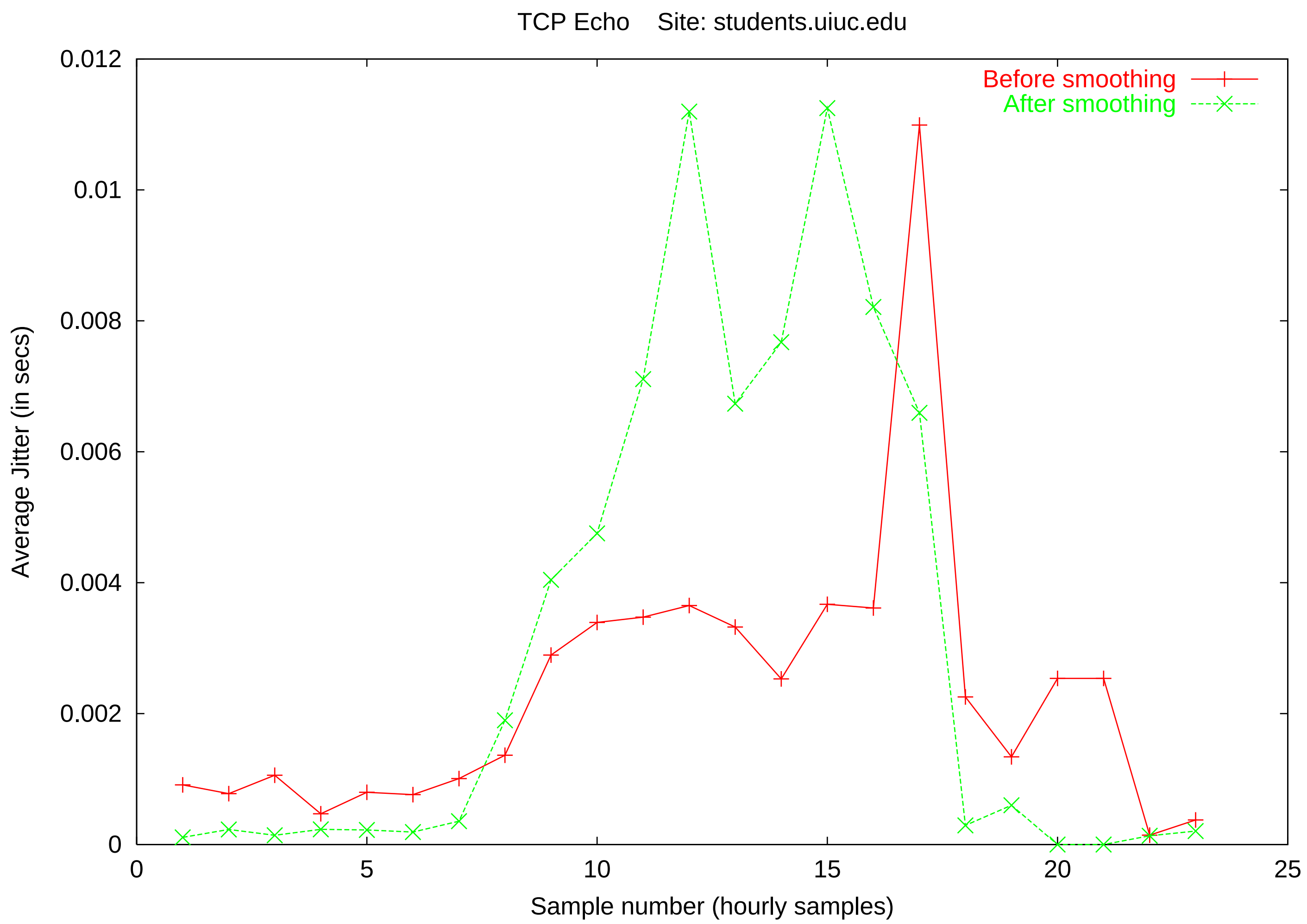
Graph 5

This is a graph similar to the previous one, but in this case the data is echoed off students.uiuc.edu which has highly variable bandwidth. Thus it could be seen that the number of packets in the buffer falls to 1 several times during the experiment. The least buffer size we see is 1, this is because only packet arrival is considered a data collection event. However this does imply that the number of packets in the buffer were zero prior to the arrival of this packet. Thus the number of times when the application had to cope with a large jitter are significantly high. In this case, smoothing is not as effective as earlier. It should also be noted that the domestic site has a smaller average jitter value(RTT) and thus also a smaller jitter buffer size estimate of 3.



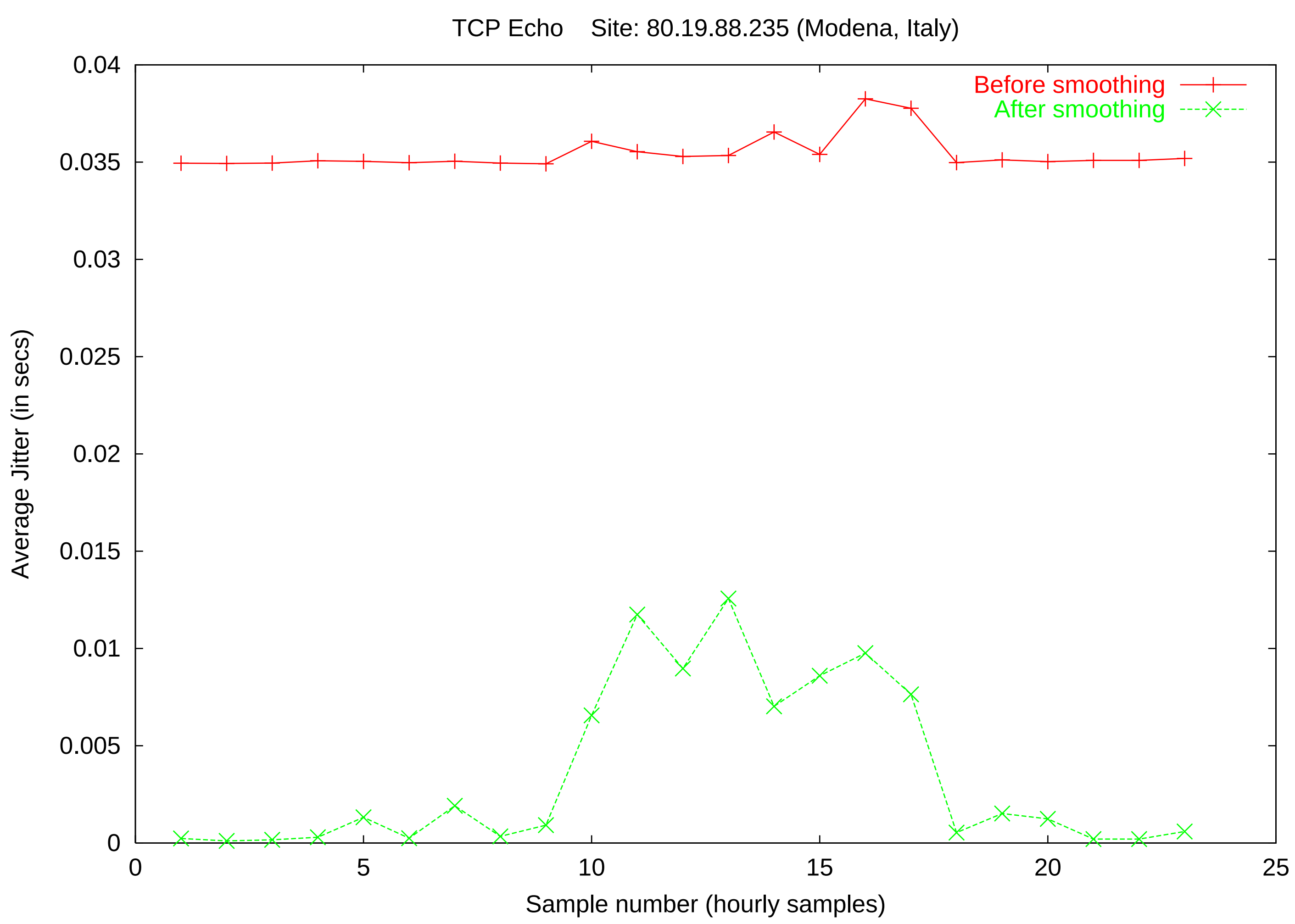
Graph 6

This graph shows results similar to those in the previous two. However the data is shown for a congested international site. The average jitter is still low, leading to a jitter buffer size estimate of about 10. However the number of instances when the buffer occupancy falls to zero are worse than the case of the site in Graph 4. Thus the number of instances when the application sees large jitters are significantly more. The buffer occupancy is also fairly stable. Compare these results with those shown in Graph 4.



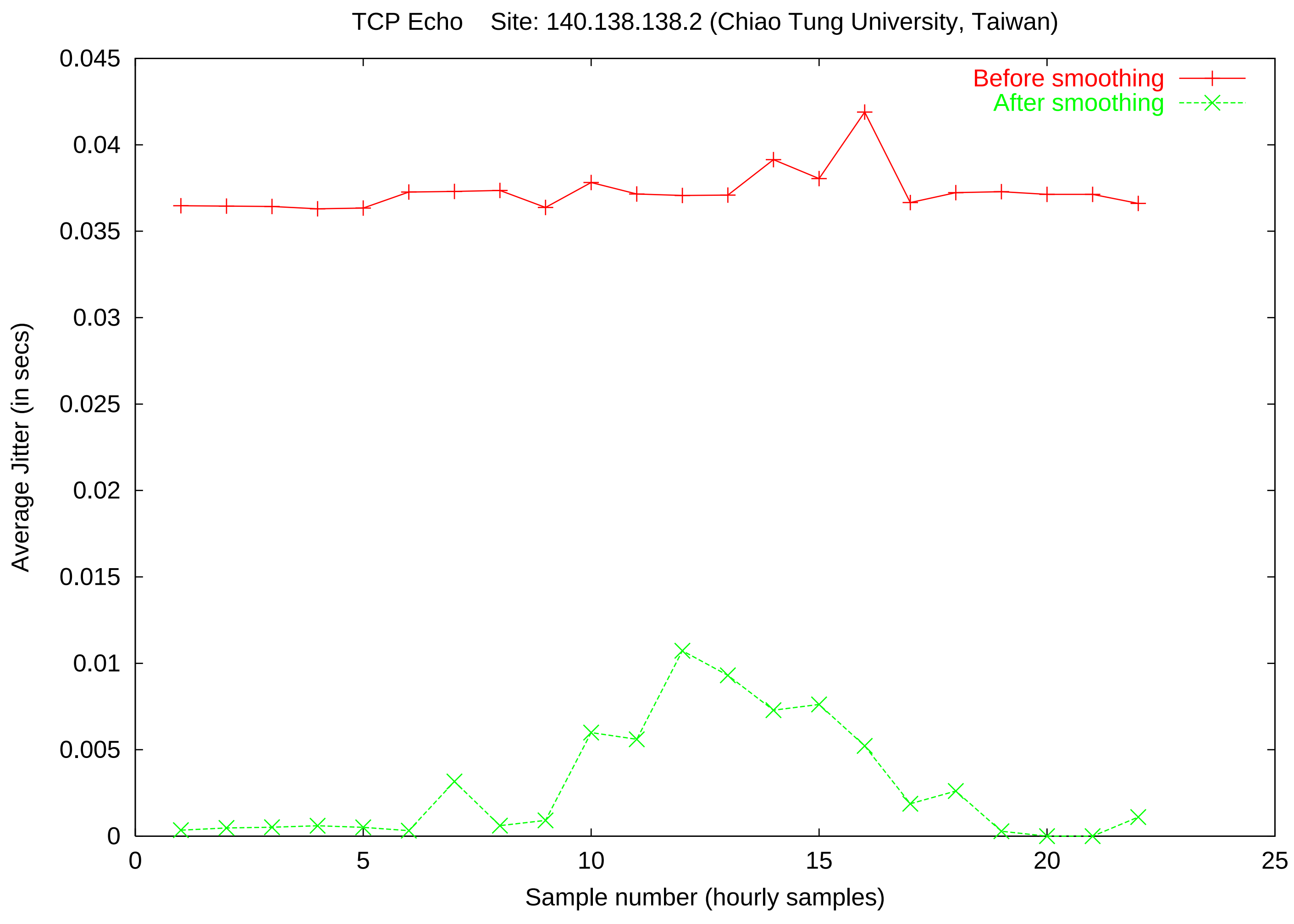
Graph 7

This graph plots the average jitters (over 5000packets) before and after smoothing on sutdents.uiuc.edu. As has been mentioned before in Graph 5, this server shows lots of transient behavior and is heavily loaded besides. Here it is seen that for most of the working period of the day, jitters after smoothing are comparable to the jitters before smoothing. Though this looks like a problem, it is not, because in the experiment the receiver sees the jitter involved in transferring data from the forwarder thread to the receiver across the LAN. Both the jitter before smoothing and that after it are very small, of the order of 10mSec. This is the minimum jitter we get across a LAN and cannot be avoided.



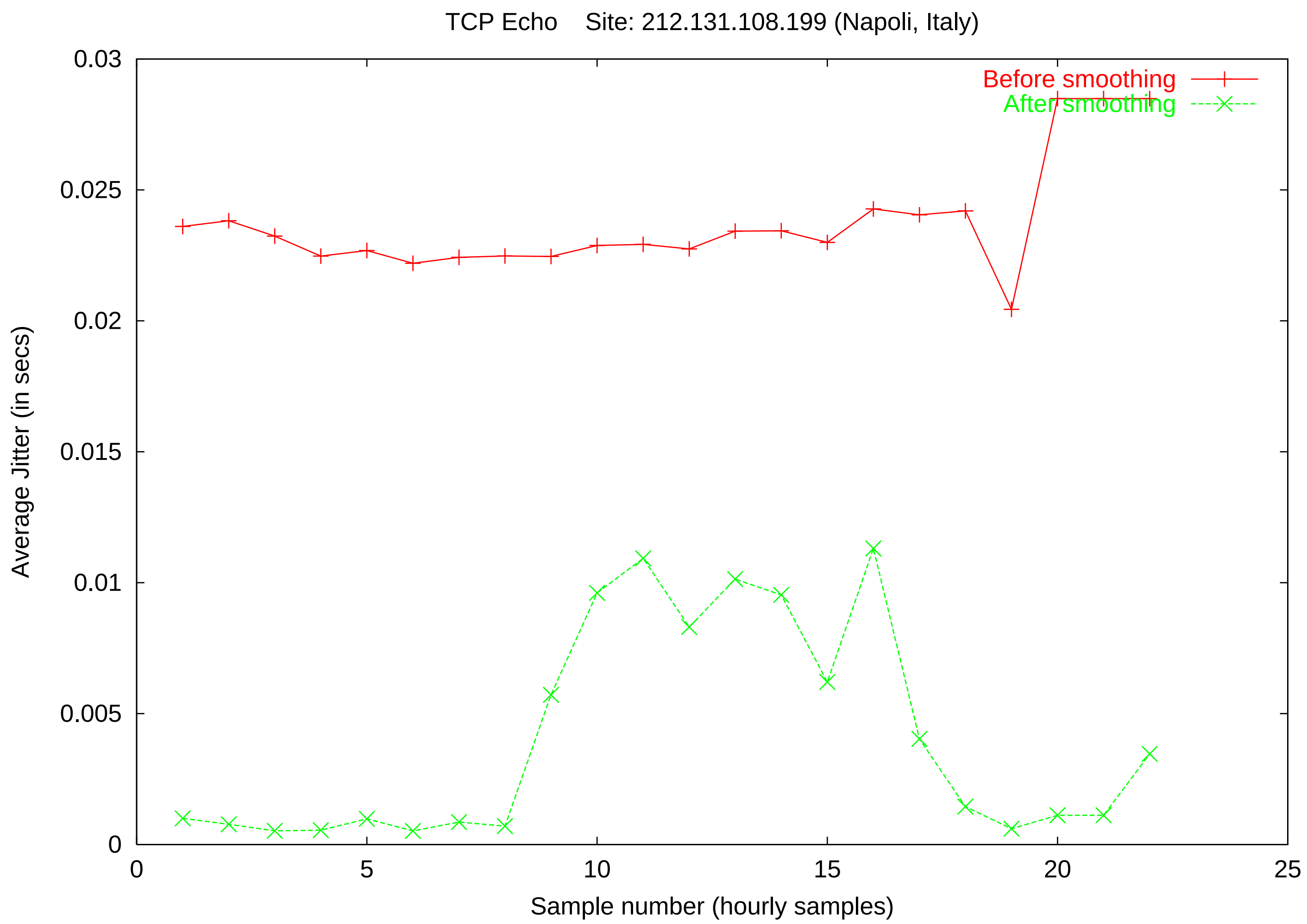
Graph 8

As in Graph 7, this plot compares the average jitter seen at various times for an international site, before and after smoothing. As has been argued in the comments after Graph 7, the jitter after smoothing is of the order of 10mSec and is entirely due to the LAN transport of data out of the jitter buffer and to the receiver. The jitter from the network is significantly larger, of the order of 35mSec and smoothing is successful.



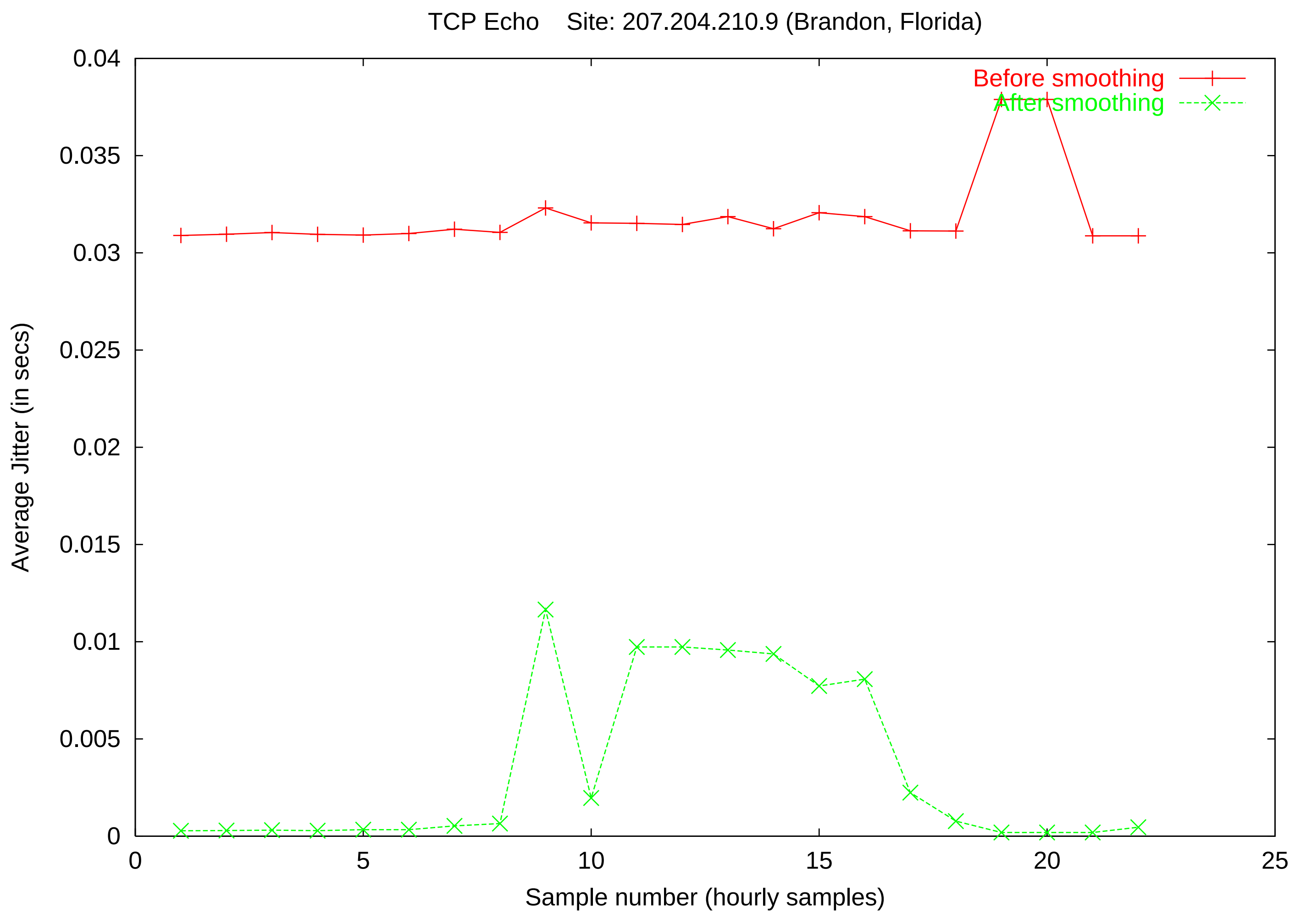
Graph 9

As in Graphs 8 and 7, this plot shows the average jitter seen at various times before and after smoothing for another international site. The jitter after smoothing is about 10mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 40mSec and has been smoothed down considerably.



Graph 10

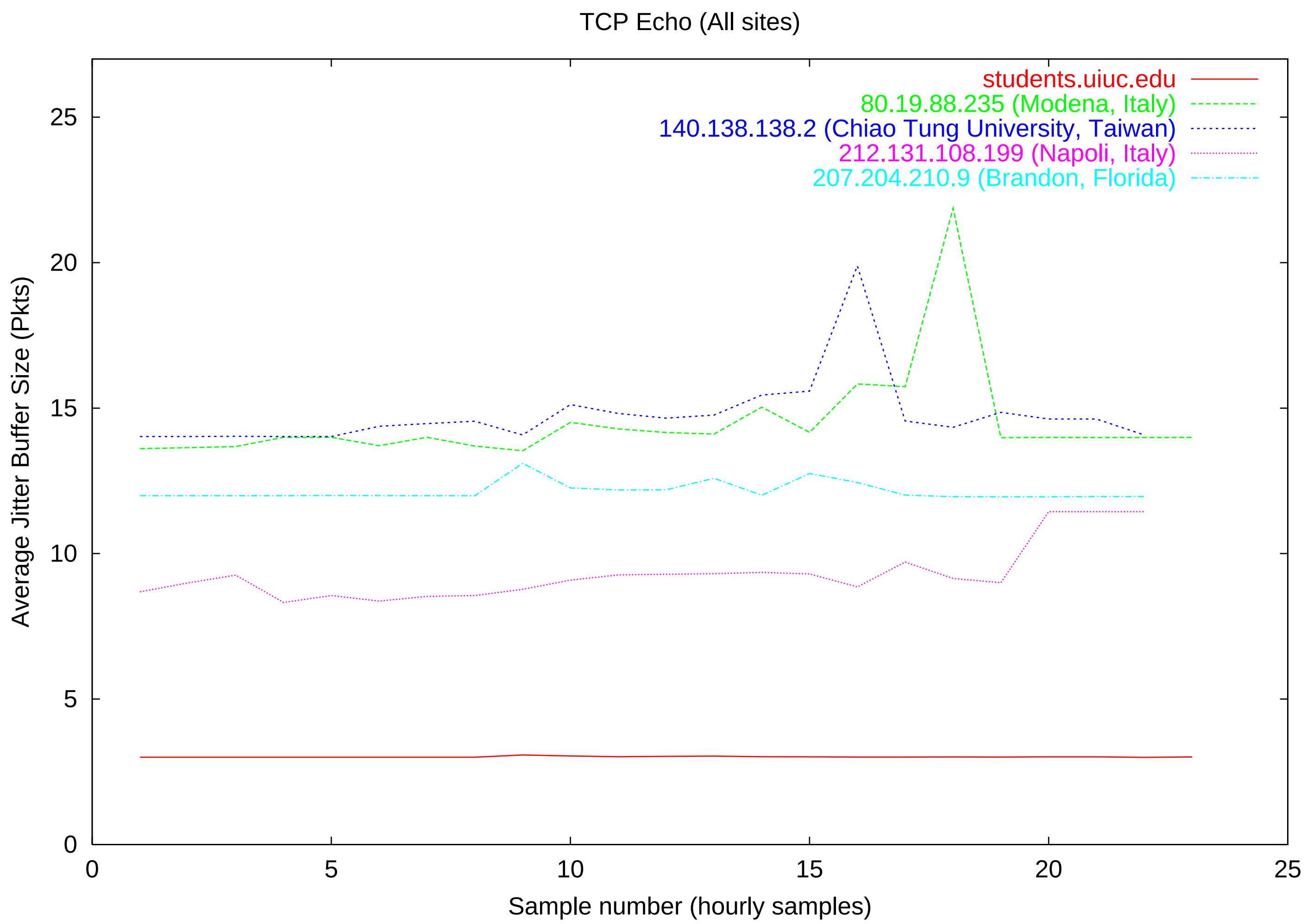
As in Graphs 8, 9 and 10, this plot shows the average jitter seen at various times before and after smoothing for another international site. The jitter after smoothing is about 10mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 25mSec and has been smoothed down considerably, (significantly smaller RTT).



Graph 11

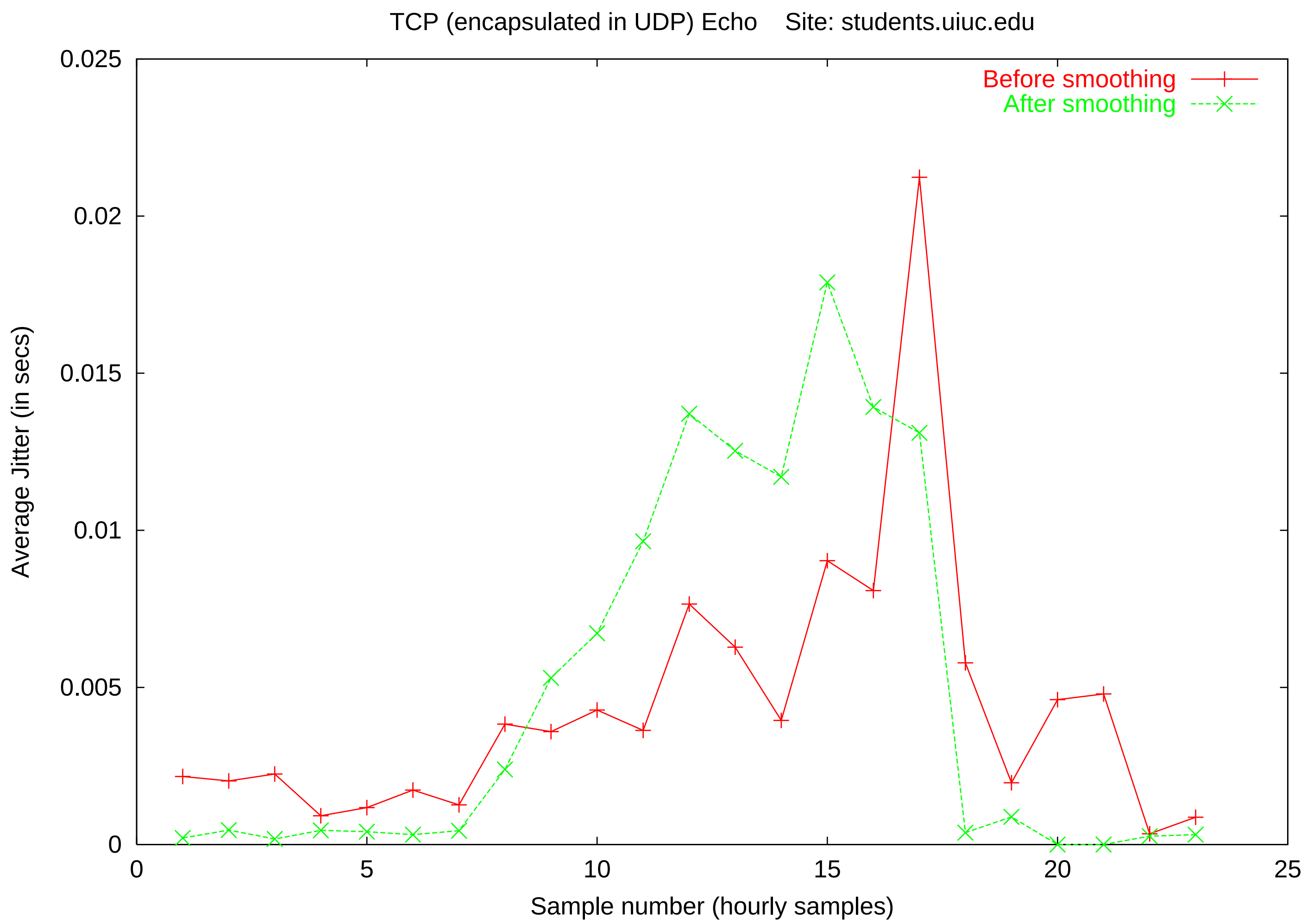
As in Graphs 8, 9, 10 and 7, this plot shows the average jitter seen at various times before and after smoothing for a domestic site in Florida. The jitter after smoothing is about 10mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 30mSec and has been smoothed down considerably.

From graphs 7 to 11, it can be safely inferred that the average jitter has been smoothed down considerably. From graph 1, it could be seen that this happens most of the time. Thus jitter is not only less on-the-average but is also less most of the time.



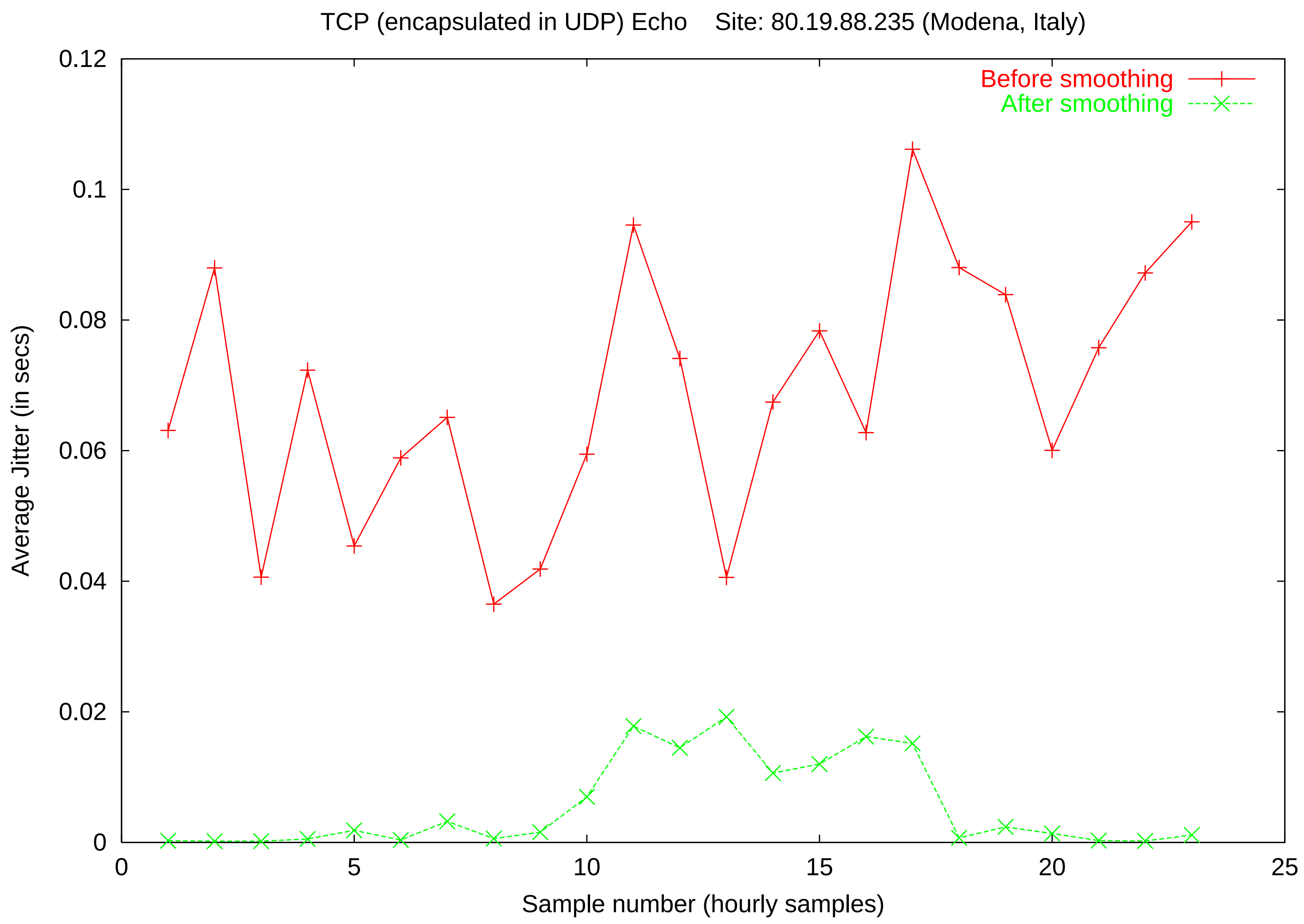
Graph 12

This graph plots the jitter buffer size estimated (average) over all samples for the five sites we experimented on. students.uiuc.edu has a significantly smaller RTT, average jitter and thereby smaller jitter buffer sizes. All the other sites have comparable RTT's and thus have jitter buffer sizes of the order of 10 to 15 packets.



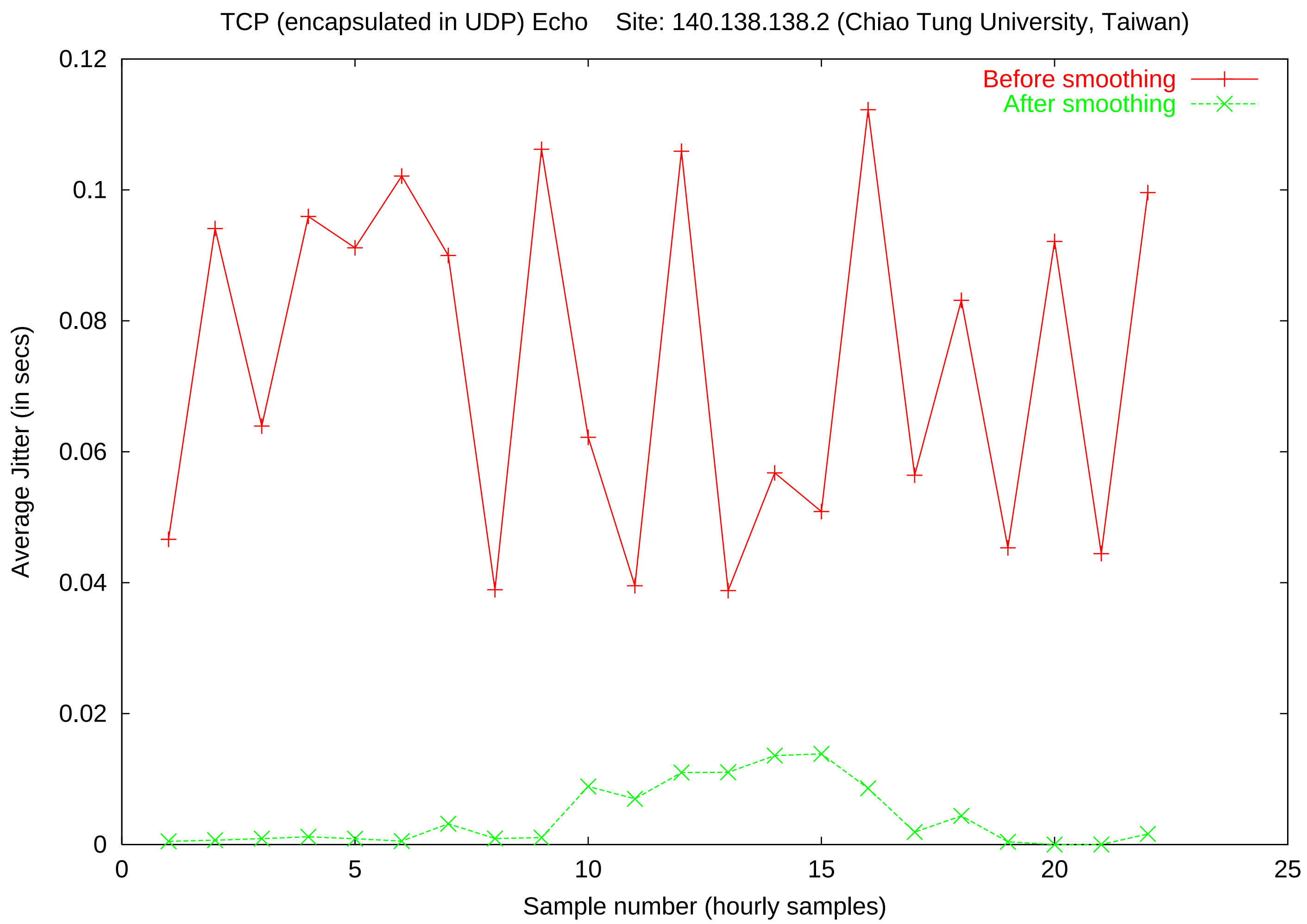
7.2 TCP Encapsulated in UDP

Graph 13 above shows the average jitters before and after smoothing when TCP packets are encapsualated in UDP and echoed off a remote UDP echo port. This plot shows the average jitter seen at various times before and after smoothing for "students.uiuc.edu". The jitter after smoothing is about 10mSec, and is almost wholly due to the local LAN introduced jitter, so is the average jitter before smoothing. When compared with the results in Graph 7 (TCP Echo for students), we see that the average jitters before smoothing are greater for TCP encapsulated in UDP while the average jitters after smoothing are roughly the same. Again, the latter is because this jitter is almost wholly due to the local LAN.



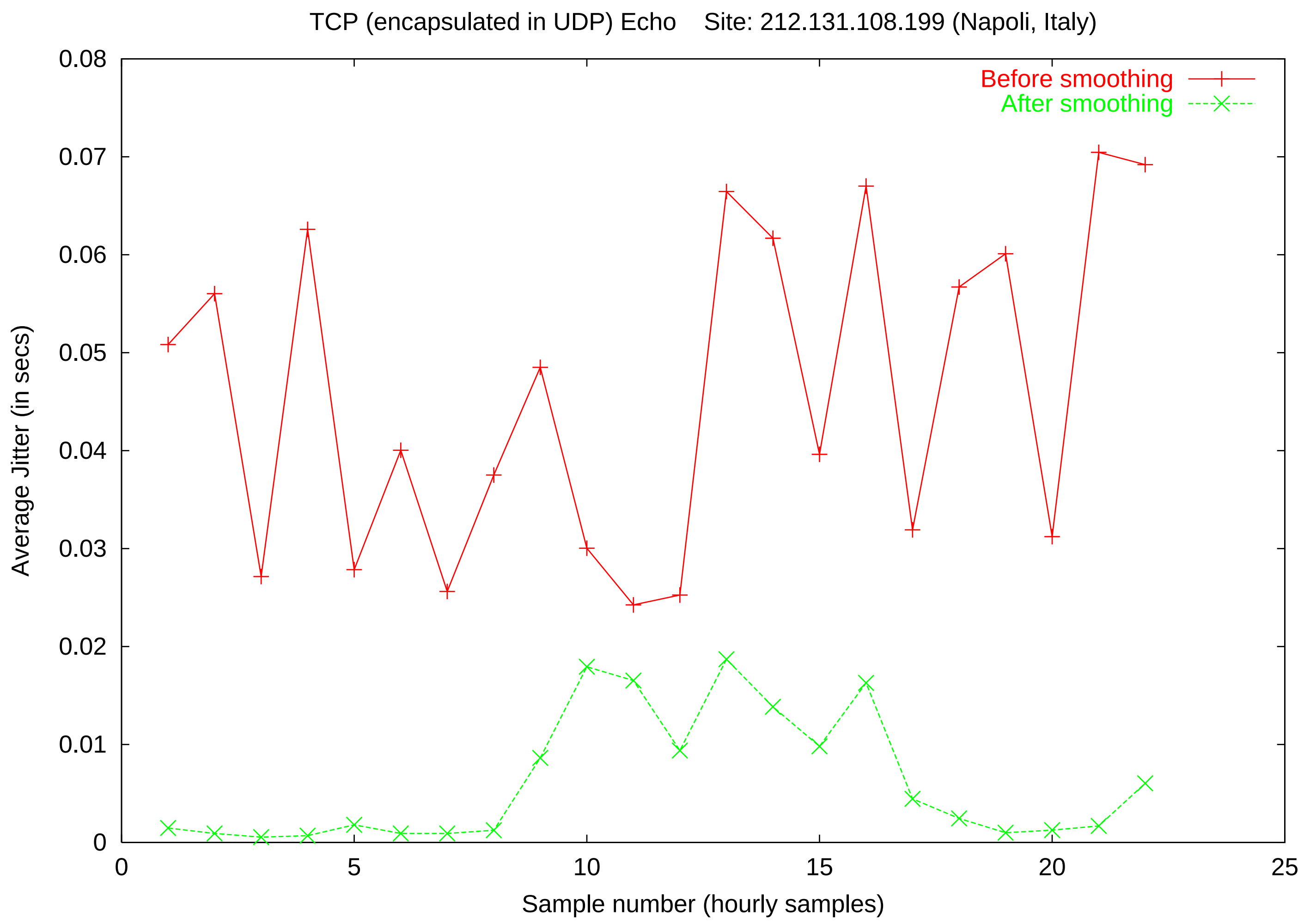
Graph 14

As in Graph 13, this graph plots the average jitter seen at various times before and after smoothing for another international site. The jitter after smoothing is about 15mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 70mSec and has been smoothed down considerably. Jitter in the TCP encapsulated over UDP scenario is slightly higher than the pure TCP case (Compare Graph 8). We conjecture that the greater variation is due to greater losses because of the use of UDP.



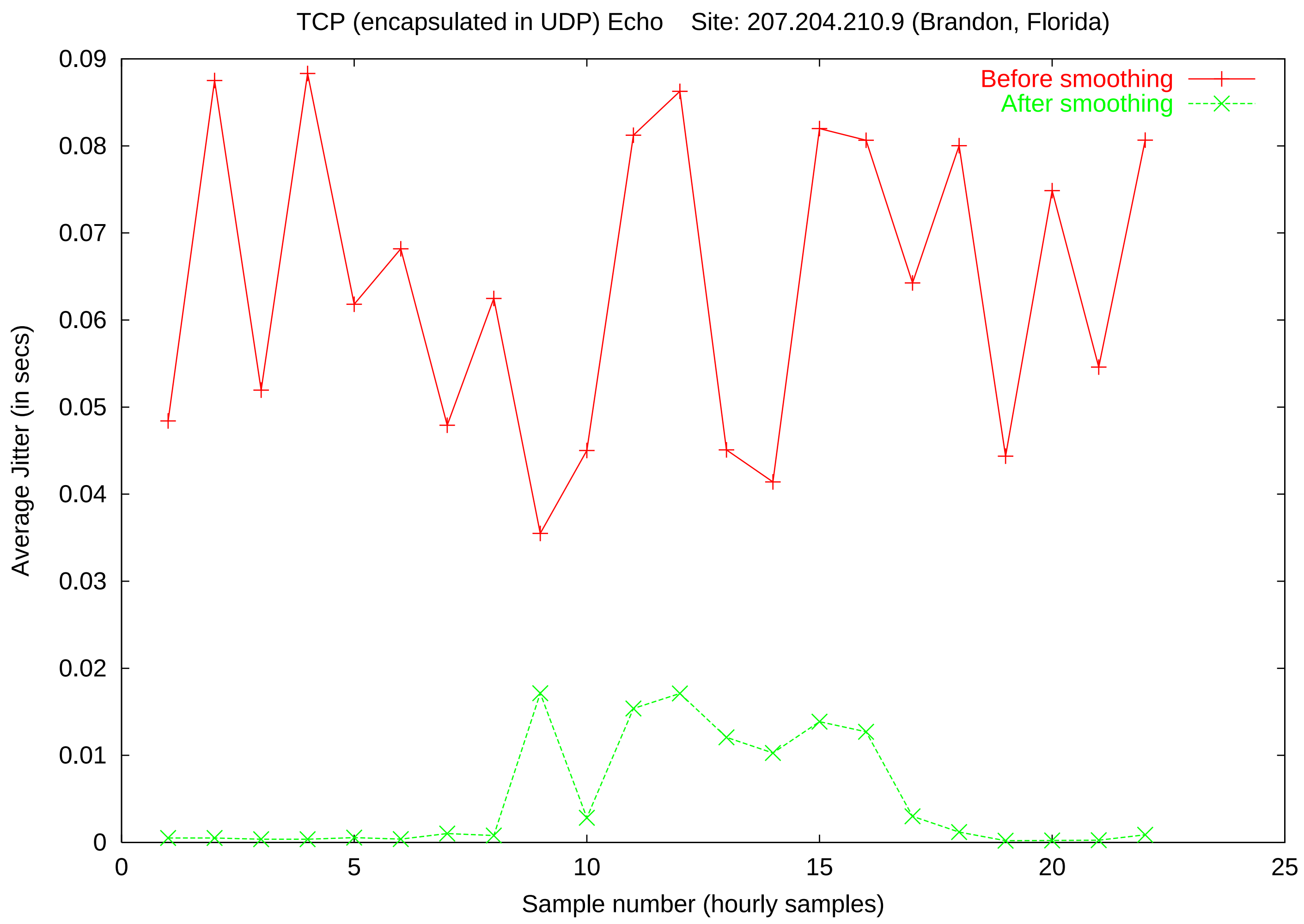
Graph 15

As in Graphs 13 and 14, this graph plots the average jitter seen at various times before and after smoothing for another international site. The jitter after smoothing is about 14mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 80mSec and has been smoothed down considerably. Jitter in the TCP encapsulated over UDP scenario is slightly higher than the pure TCP case (Compare Graph 9).



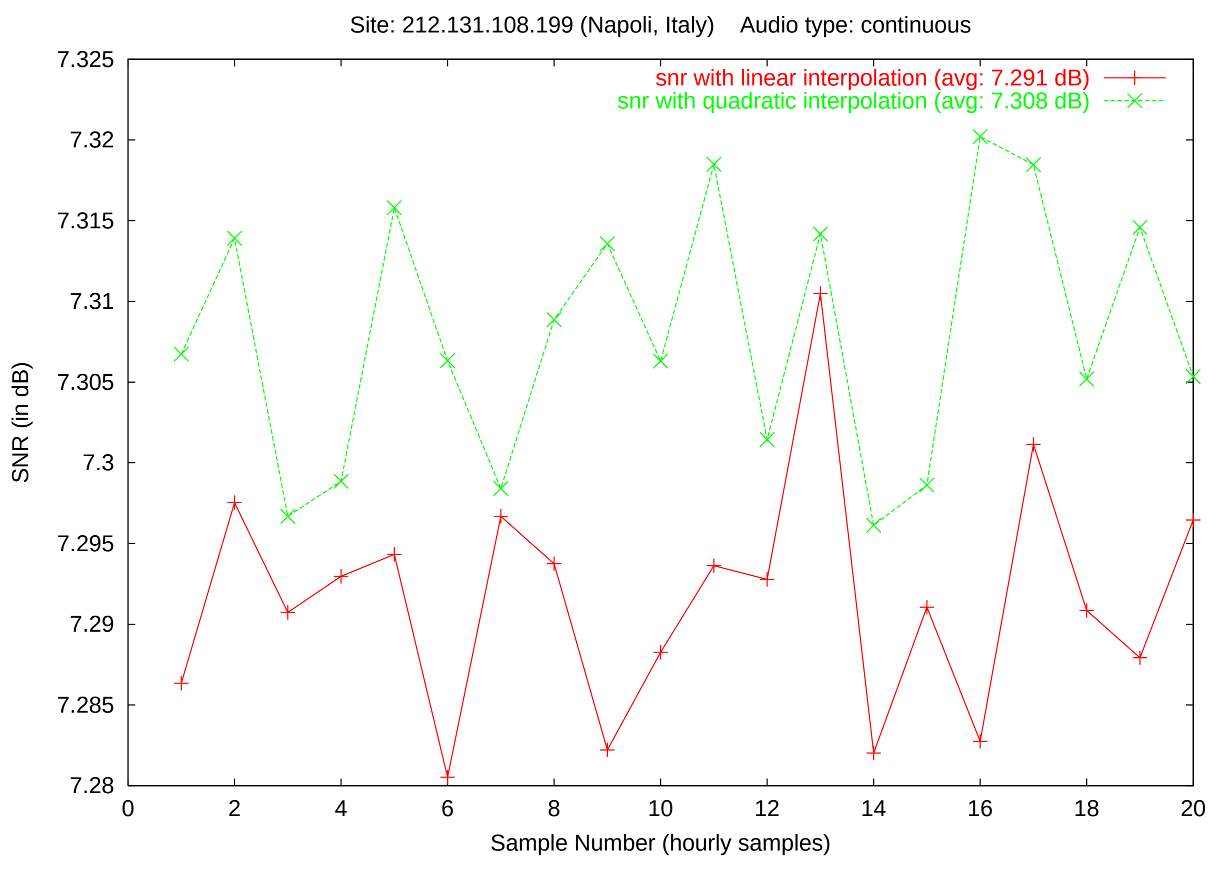
Graph 16

As in Graphs 13, 14 and 15, this graph plots the average jitter seen at various times before and after smoothing for another international site. The jitter after smoothing is about 16mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 50mSec and has been smoothed down considerably. Jitter in the TCP encapsulated over UDP scenario is slightly higher than the pure TCP case (Compare Graph 10).



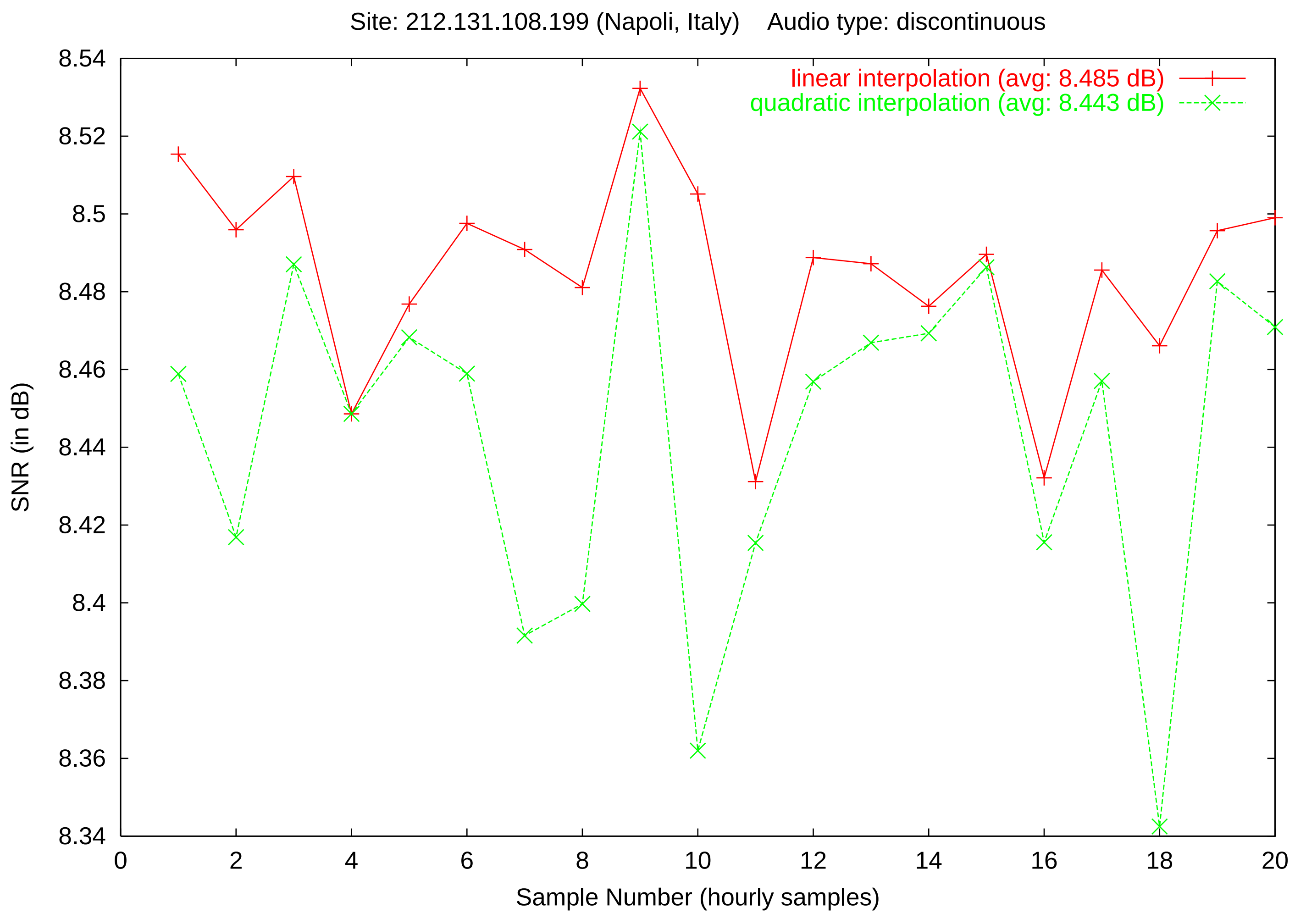
Graph 17

As in Graph 13 to 16, this graph plots the average jitter seen at various times before and after smoothing for another domestic site. The jitter after smoothing is about 15mSec, and is almost wholly due to the local LAN introduced jitter. Jitter before the network is of the order of 60mSec and has been smoothed down considerably. Jitter in the TCP encapsulated over UDP scenario is slightly higher than the pure TCP case (Compare Graph 11).



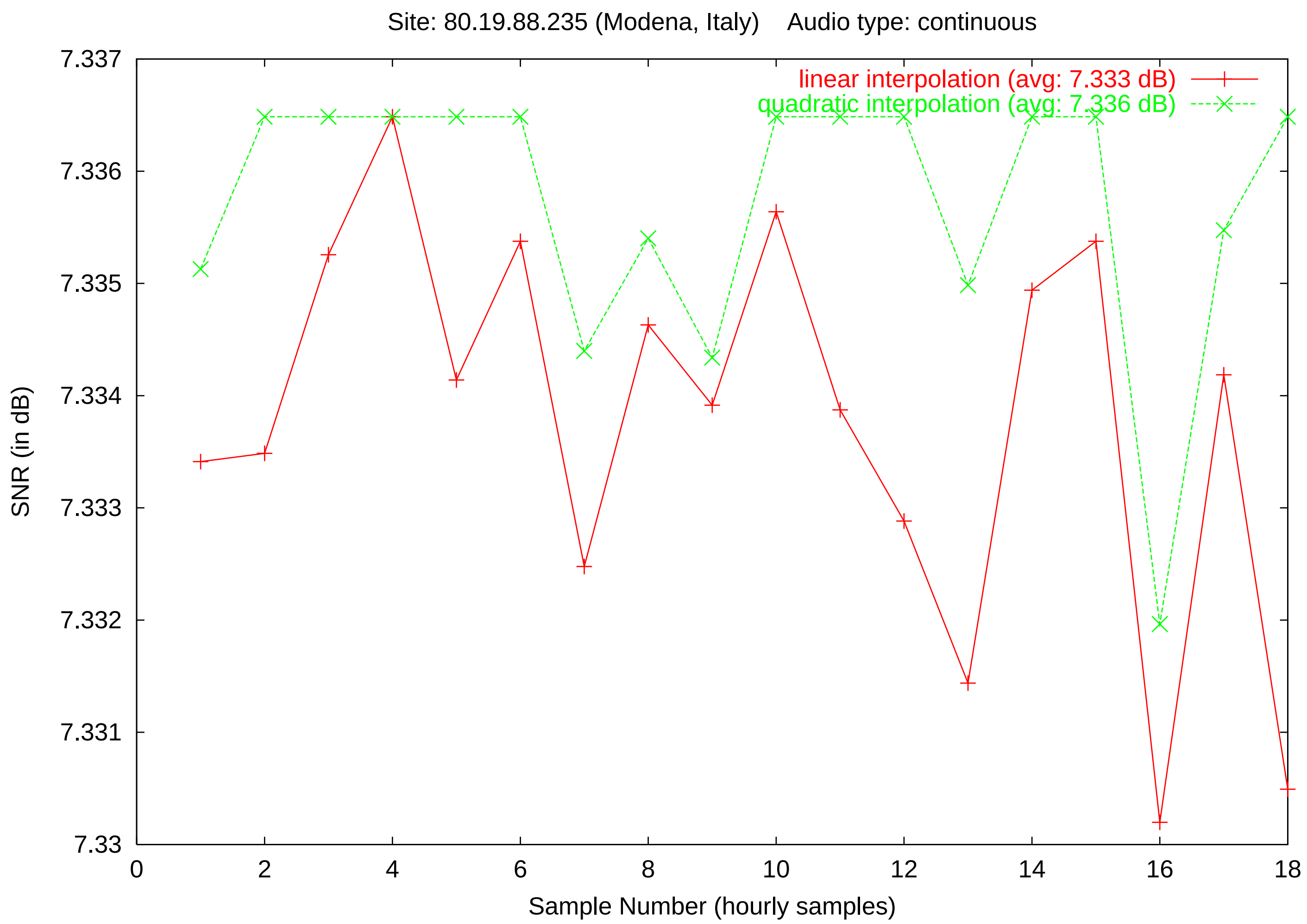
7.3 UDP Echo

Graph 18 above shows the variation of SNR calculated for each hourly sample. This data is for an international site in Napoli, Italy. The base SNR of the file, the value obtained by just encoding and decoding the voice sample without any transmissions over the internet, was about 7.6. The voice sample transmitted was the one with smooth variations (no sharp changes or lack of differentiability like in a piecewise linear curve). Hence, quadratic interpolation, which fits a smoother quadratic curve on the missing samples, is expected to perform better—which is precisely what we observe above. Linear interpolation tries to fit a piecewise linear curve and hence performs (relatively) poorly. There is also a considerable variation in SNR over the 24 hour period as this site was an international site and losses (shown in Graph 28) varied throughout the period. The correlation between loss fraction and SNR is quite clear on looking at this graph and Graph 28—a dip in SNR has a corresponding spike in loss fraction. 4-way interleaving was used.



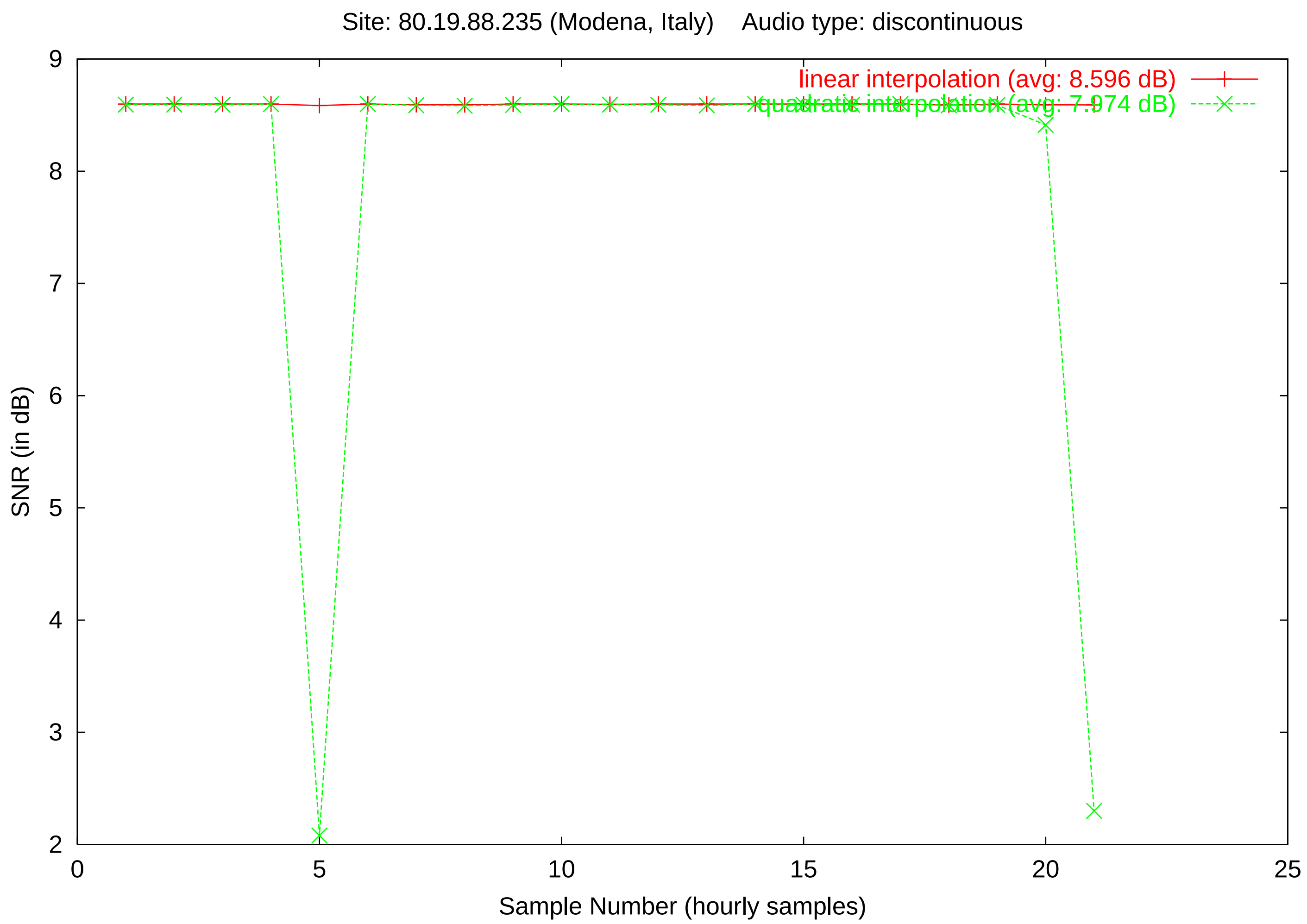
Graph 19

Graph 19 above shows the variation of SNR for the same site as the previous graph (Graph 18), but this time the audio sent was the one with a voice counting numbers—no smooth variations but more like a piecewise linear curve. Hence, one would expect linear interpolation to give superior results and the exprimental results above justify it. The base SNR for this audio stream was about 9.12. Again, the correlation with Graph 28 can be made. Again, 4-way interleaving was used.



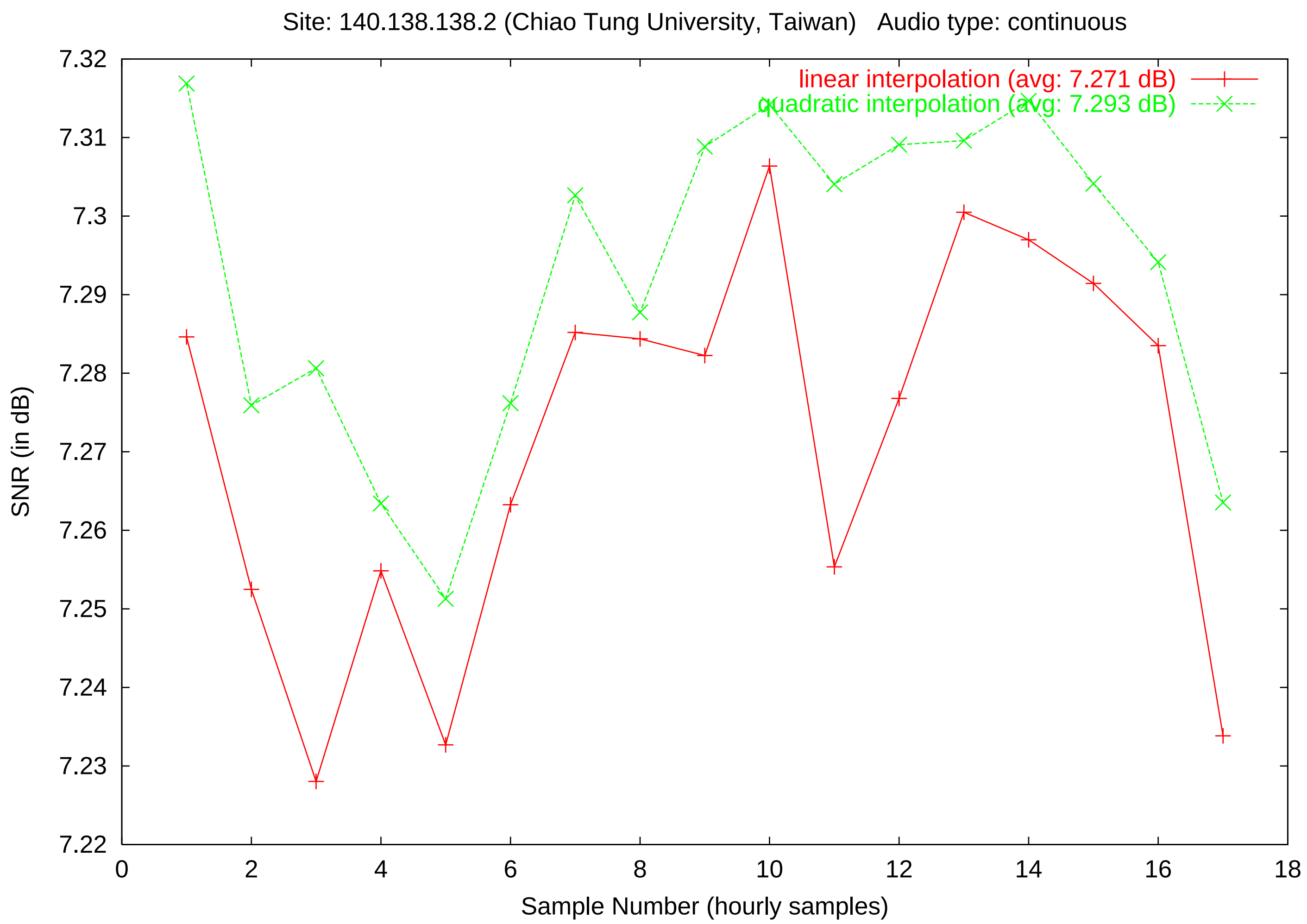
Graph 20

Graph 20 above shows the variation of SNR for an international site in Modena, Italy. As in Graph 18, the voice sample is the one with smooth variations (base SNR 7.6), and as explained for Graph 18, quadratic interpolation performs better. Graph 29 shows the corresponding loss fraction and a clear one-to-one correspondence can be made between losses and dips in SNR values.



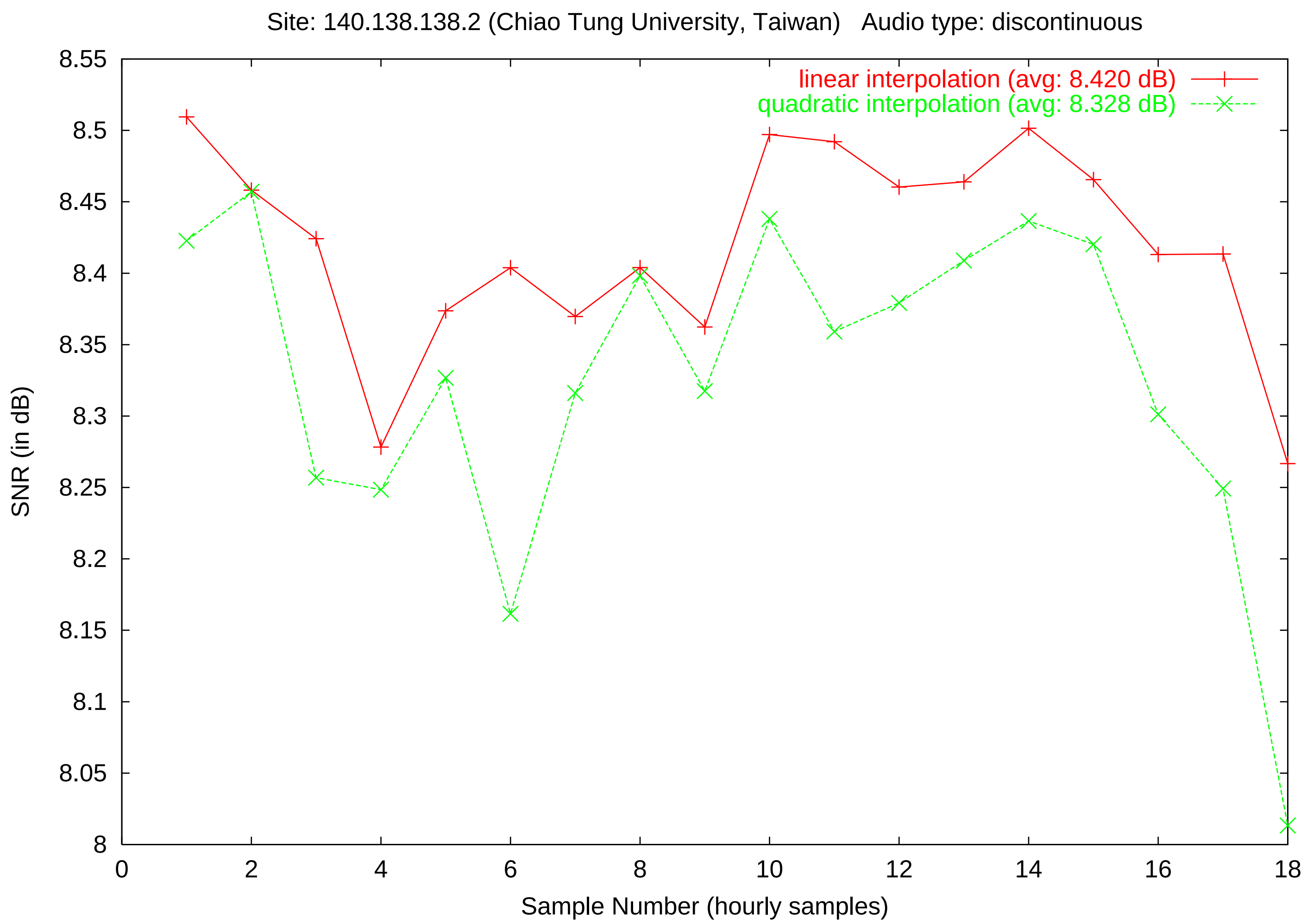
Graph 21

Graph 21 above shows the SNR variation was the same site (Modena, Italy), but this time with discontinuous (voice counting numbers) audio stream. As expected and explained for Graph 19, linear interpolation performs better than quadratic interpolation. The base SNR was about 9.12. The corresponding loss file is shown in Graph 29. The site exhibited few very steep losses while sending the stream and using quadratic interpolation as is evident from the sharp drop in SNR and a corresponding spike in Graph 29. This loss hides the smaller variations and makes the rest of the graph quite consistent.



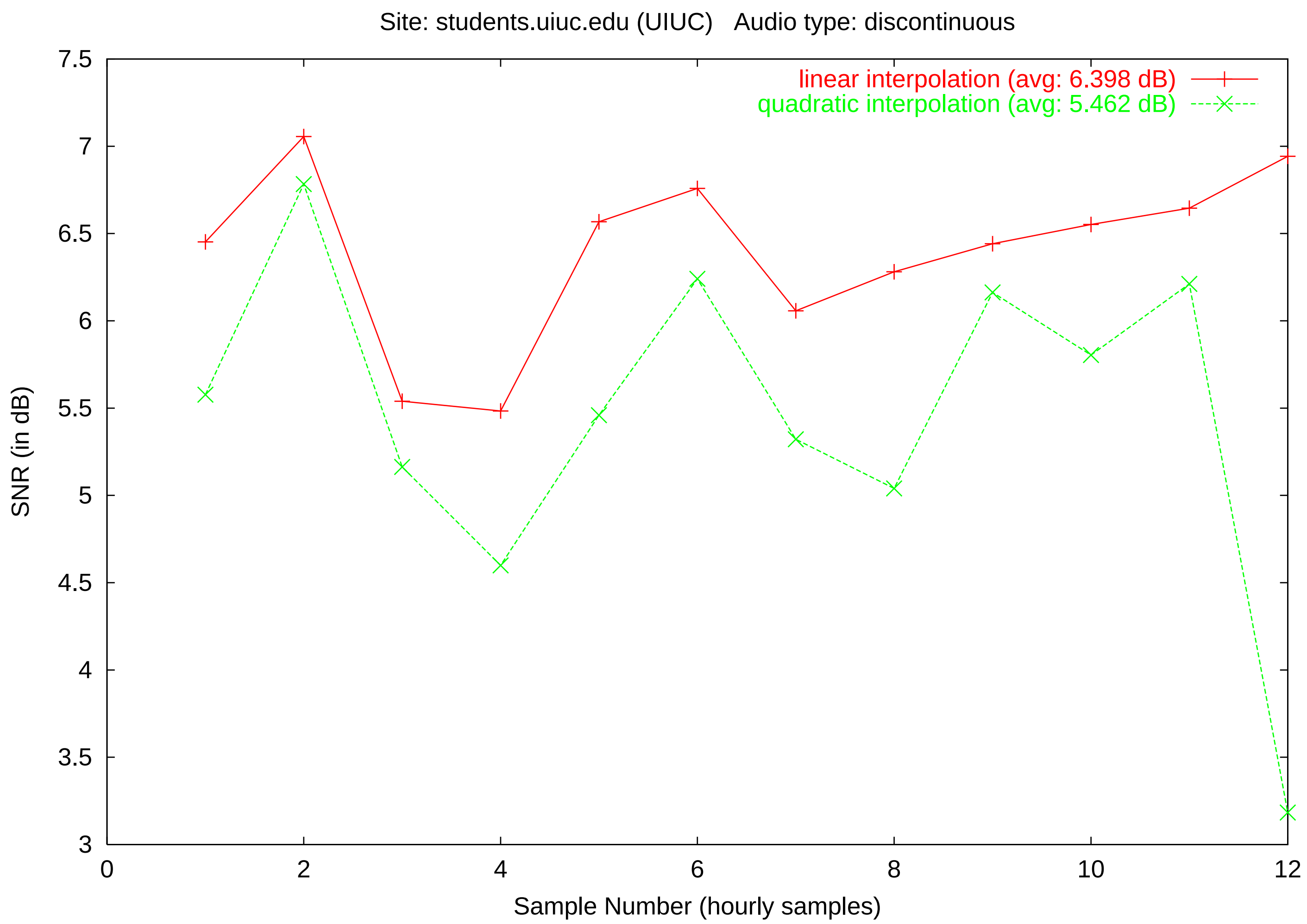
Graph 22

Graph 22 above shows the variation of SNR for an international site at Chiao Tung University in Taiwan. The voice sample is the one with smooth variations (base SNR 7.6), and as explained for Graph 18 and 20, quadratic interpolation performs better. Graph 30 shows the corresponding loss fraction and a clear one-to-one correspondence can be made between losses and dips in SNR values. On the whole the site is reasonably good (avg SNR close to base SNR)



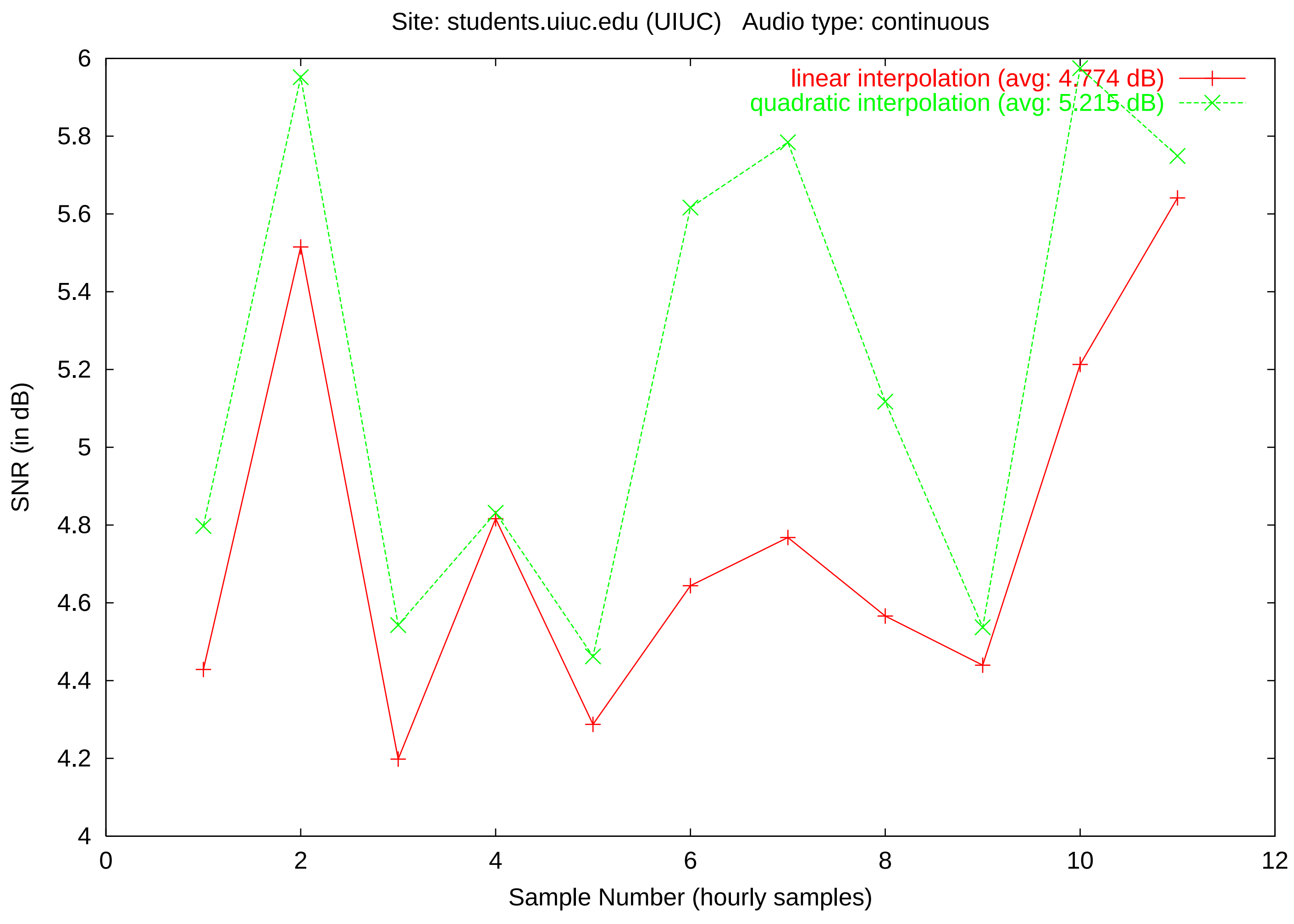
Graph 23

Graph 23 above shows the SNR variation for the same site (Chiao Tung University in Taiwan), but this time with discontinuous (voice counting numbers) audio stream. As expected and explained for Graph 19 and 21, linear interpolation performs better than quadratic interpolation. The base SNR was about 9.12. The corresponding loss file is shown in Graph 30. On the whole the amount of losses wasn't very high and the site shows a reasonably consistent SNR (all values between 8.0 and 8.55).



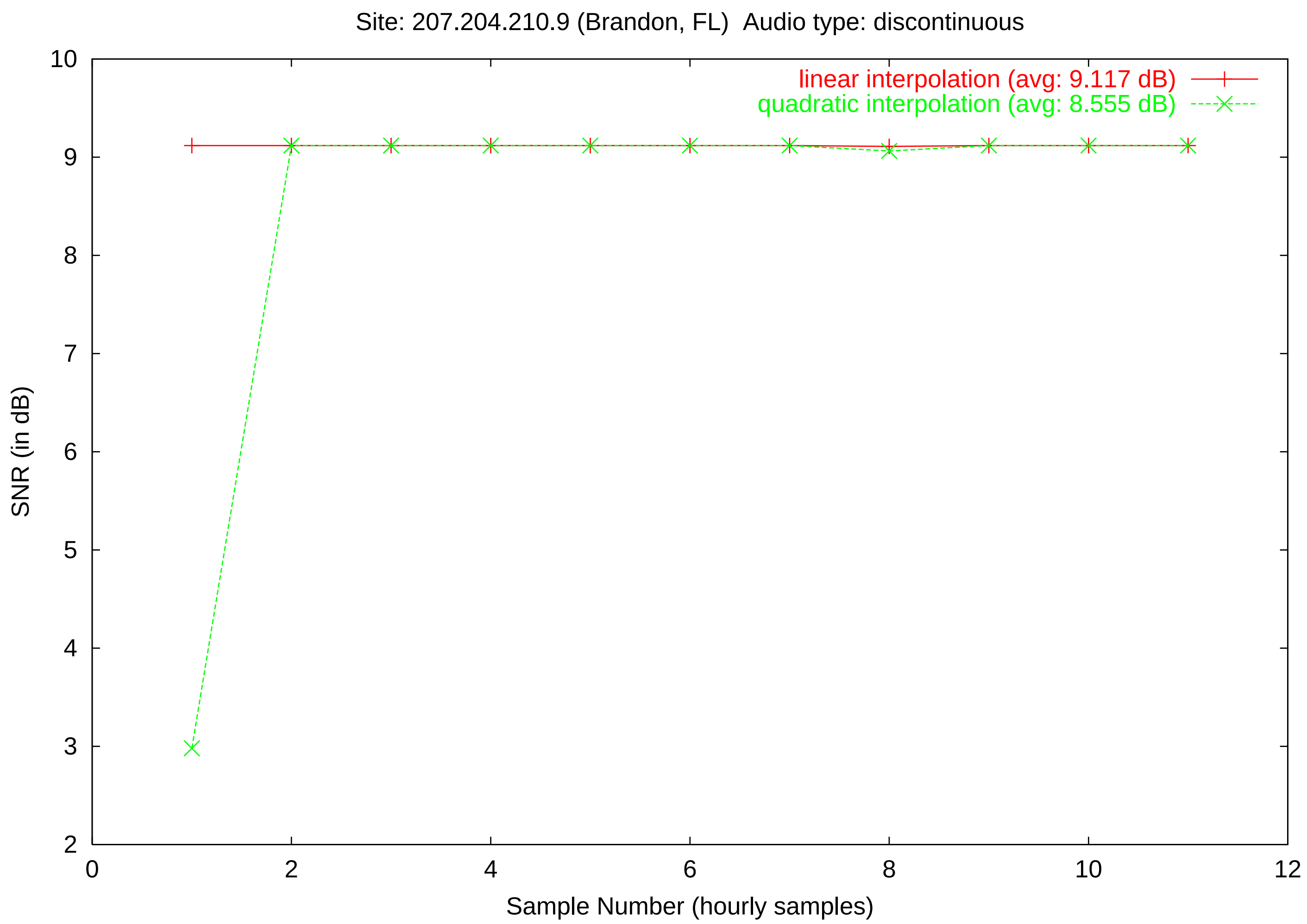
Graph 24

Graph 24 above shows the SNR variation for a domestic site:students.uiuc.edu at UIUC, with discontinuous (voice counting numbers) audio stream. As expected and explained for Graph 19, linear interpolation performs better than quadratic interpolation. The base SNR was about 9.12. Graph 31 shows the corresponding loss fractions. This site is a rather "rowdy" site, in that despite being a local site, it suddenly drops a lot of UDP echo packets. Hence the huge variation in SNR values. This is also shown in large variations (and large values) of loss fraction in Graph 21. Two way interleaving was used here.



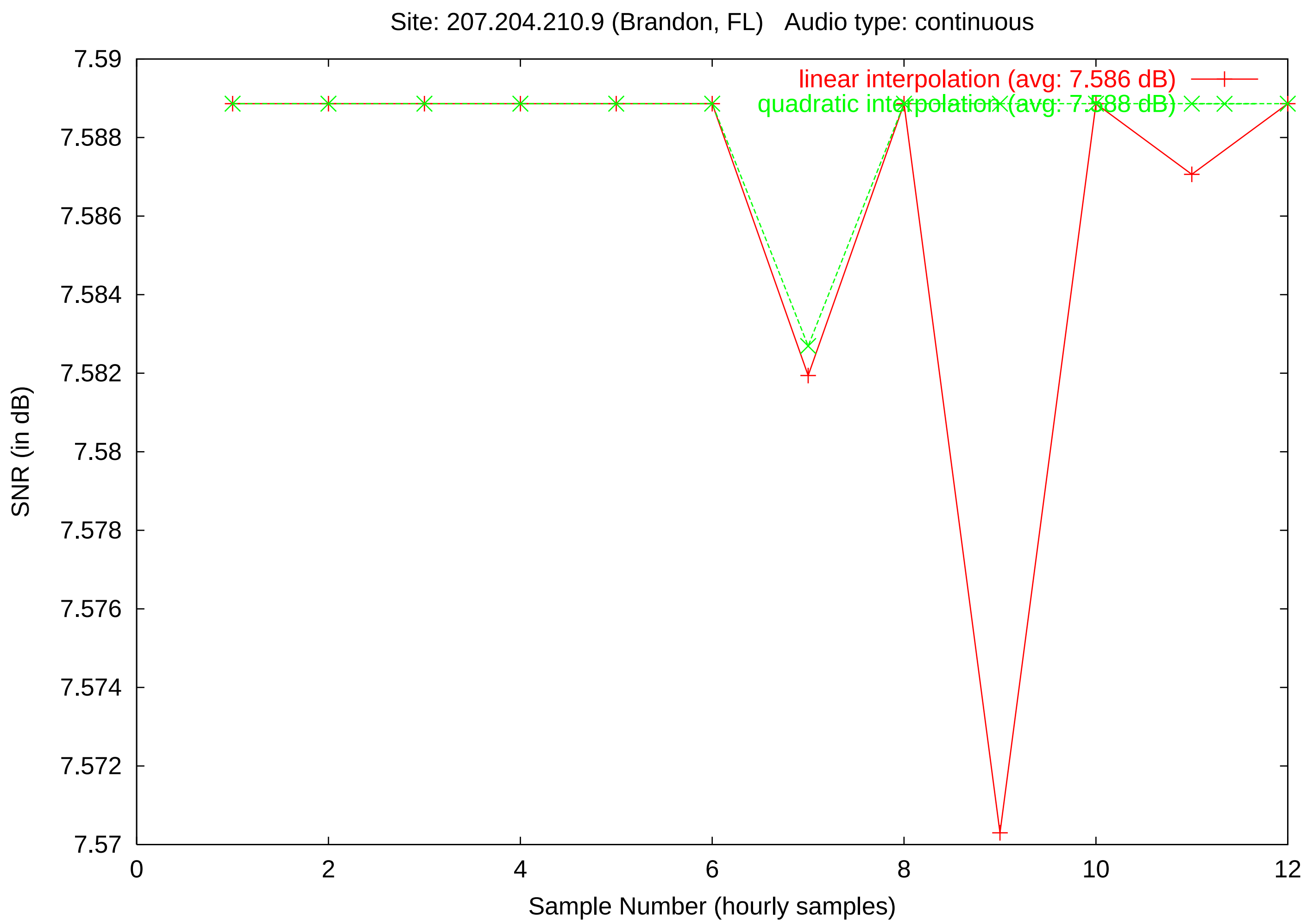
Graph 25

Graph 25 above shows the variation of SNR for a domestic site: students.uiuc.edu at UIUC. The voice sample is the one with smooth variations (base SNR 7.6), and as explained for Graph 18, 20 and 22, quadratic interpolation performs better. Again, due to large losses at this site, the average SNR is low and there is a huge amount of variation.



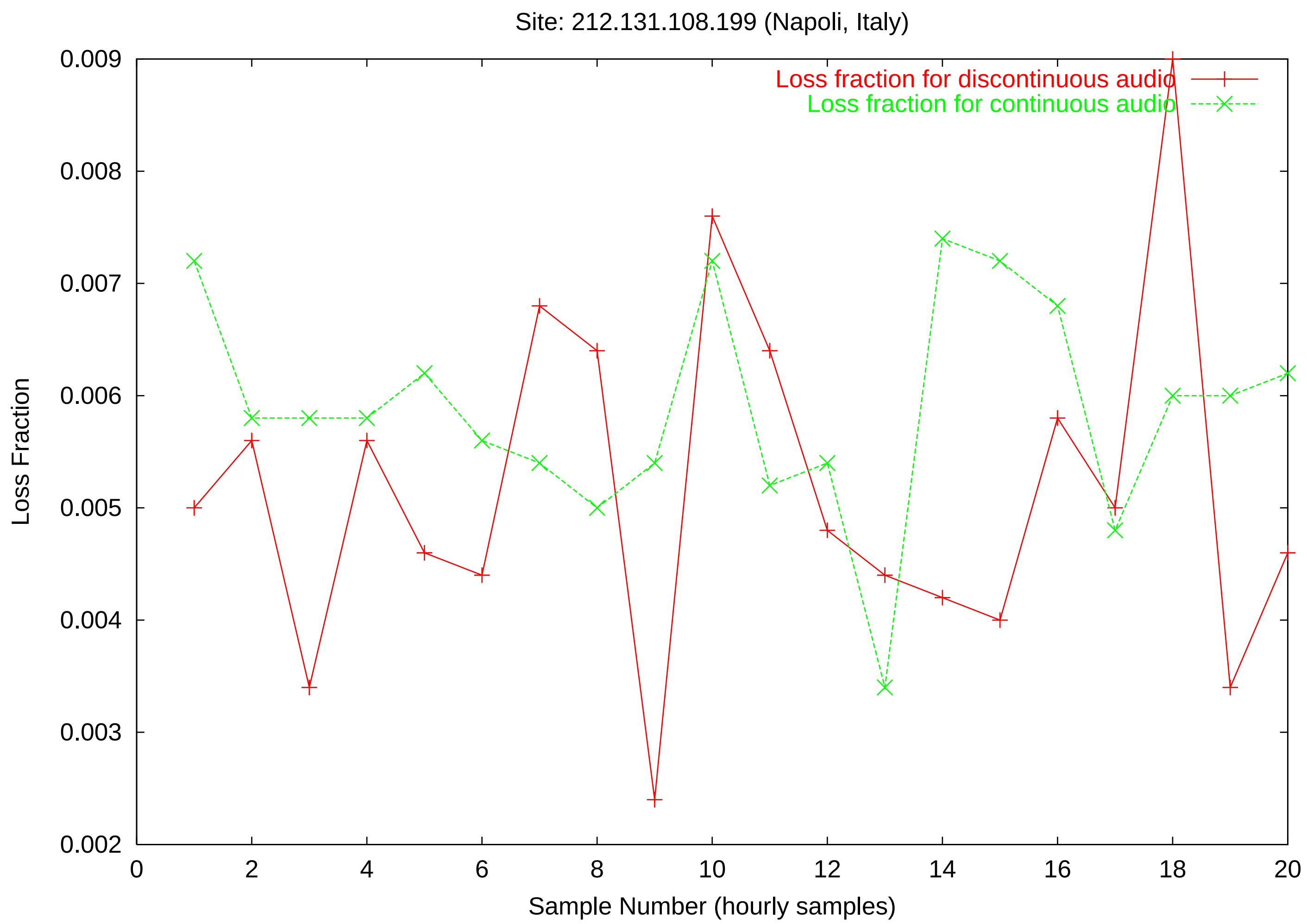
Graph 26

Graph 26 above shows the SNR variation for a domestic site at Brandon, Florida, using discontinuous (voice counting numbers) audio stream. As expected and explained for Graph 19, linear interpolation performs better than quadratic interpolation. The base SNR was about 9.12. Graph 32 shows the corresponding loss fractions. This site performs very well, except for one anomalous value. Hence the avg SNR is very high and quite consistent too.



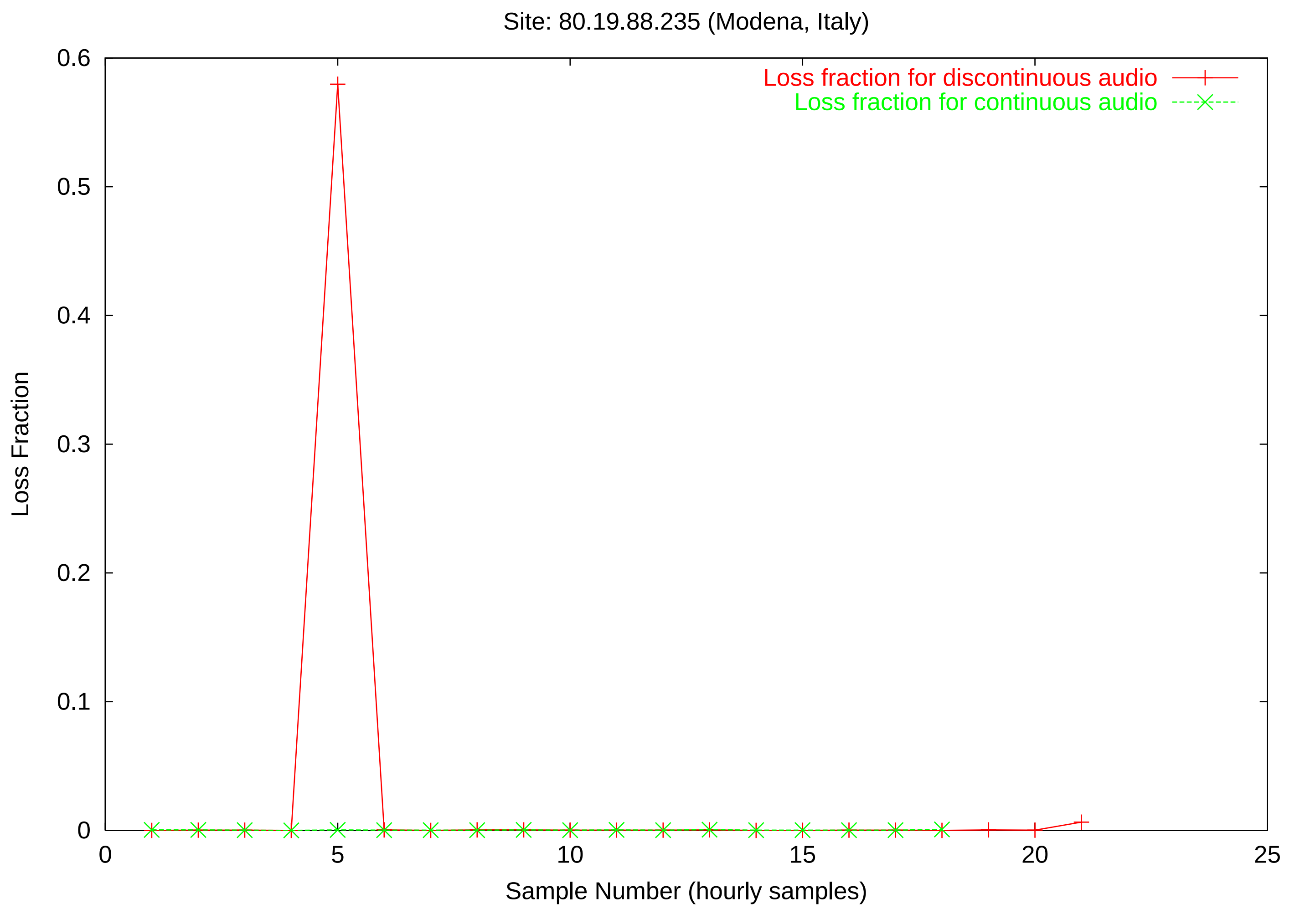
Graph 27

Graph 27 above shows the variation of SNR for the same site in Brandon, FL. The voice sample is the one with smooth variations (base SNR 7.6), and as explained for Graph 18, 20 and 22, quadratic interpolation performs better. Again, due to fewes losses, the avg SNR is quite close to the base value and reasonably consistent.



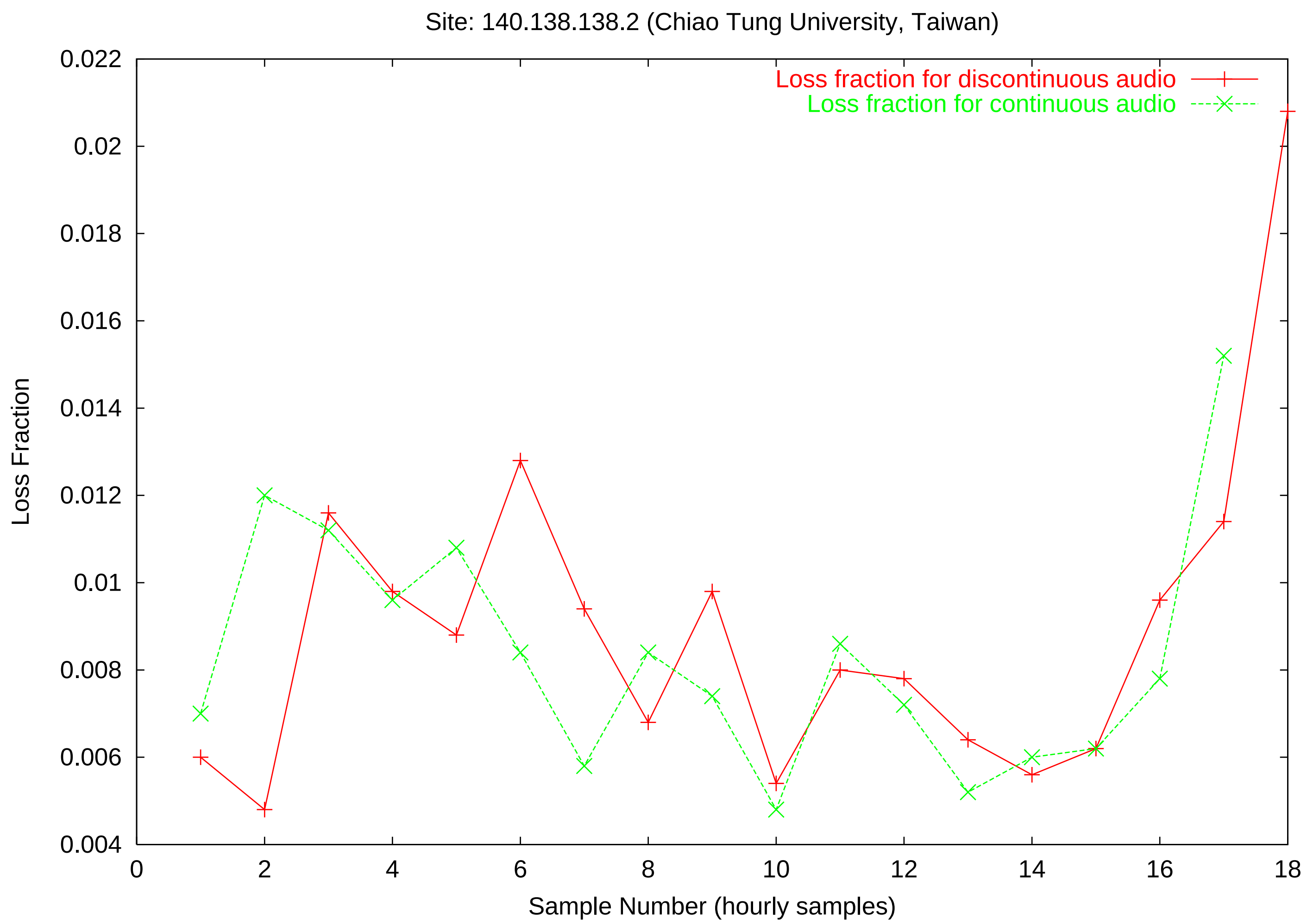
Graph 28

Graph 28 above shows the loss fraction for an international site in Napoli, Italy, both while transmitting the continuos audio stream and the discontinuous audio stream. 4-way interleaving was used. This is the total loss fraction, i.e. the number of packets lost (and not the number of audio samples that couldn't be regenerated). The loss varies between 0.2 and 0.9 percent.



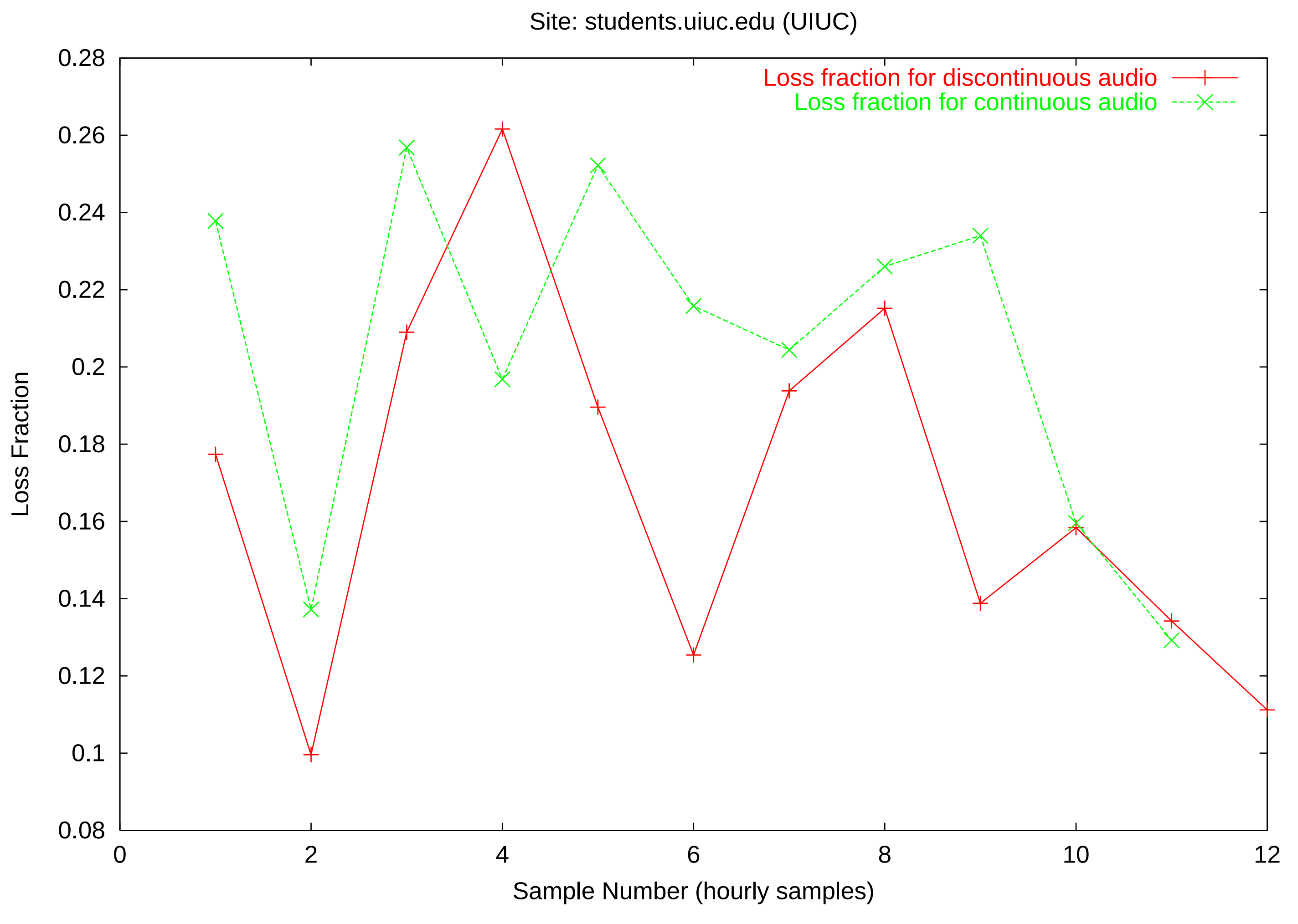
Graph 29

Graph 29 above shows the loss fraction for an international site in Modena, Italy, both while transmitting the continuos audio stream and the discontinuous audio stream. 4-way interleaving was used. This is the total loss fraction, i.e. the number of packets lost (and not the number of audio samples that couldn't be regenerated). There is one big spike and the other values are more-or-less small.



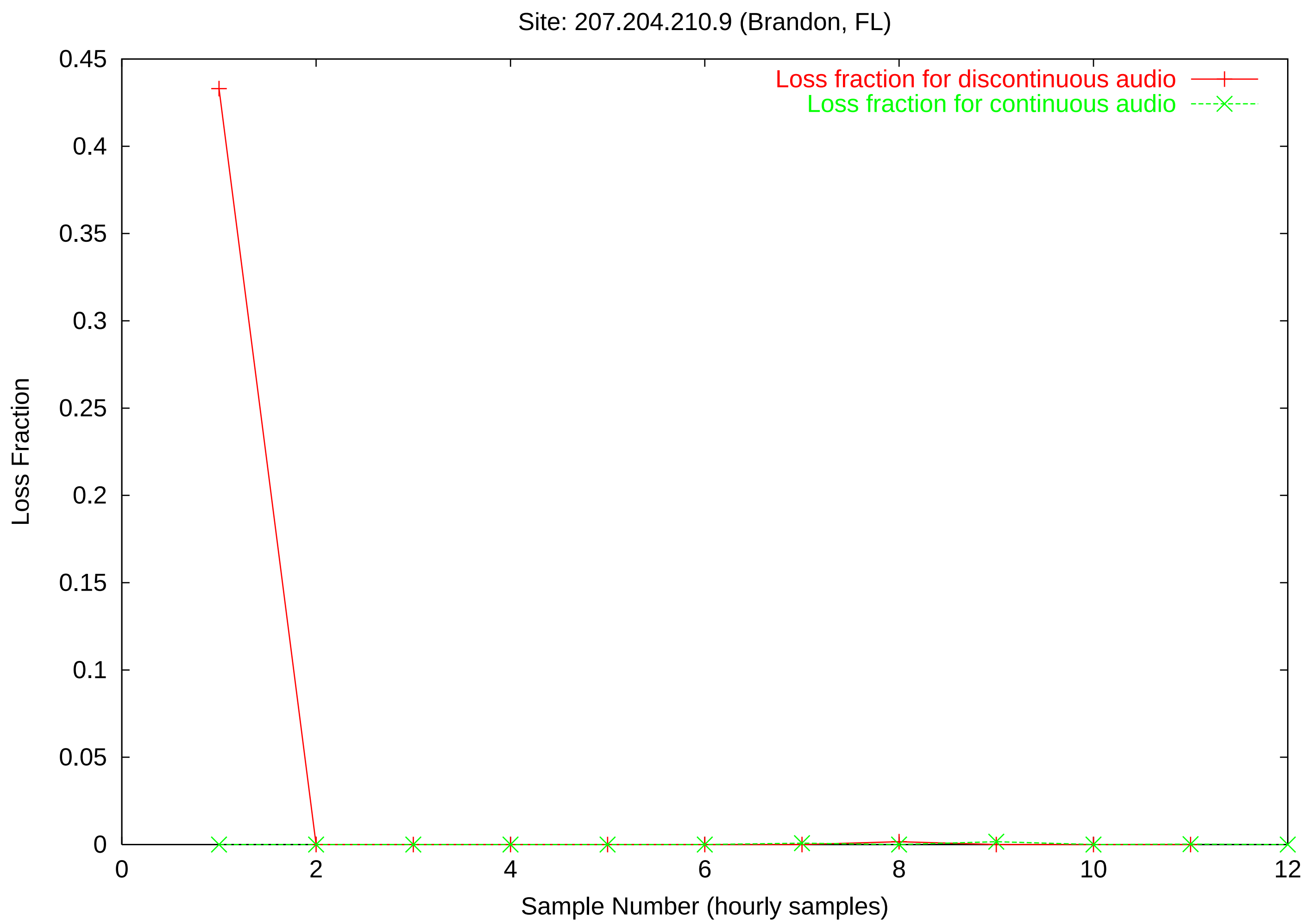
Graph 30

Graph 30 above shows the loss fraction for an international site in Taiwan, both while transmitting the continuos audio stream and the discontinuous audio stream. 4-way interleaving was used. This is the total loss fraction, i.e. the number of packets lost (and not the number of audio samples that couldn't be regenerated). The loss varies between 0.4 and 2.2 percent.



Graph 31

Graph 31 above shows the loss fraction for students.uiuc.edu, a local site, both while transmitting the continuos audio stream and the discontinuous audio stream. 2-way interleaving was used. This is the total loss fraction, i.e. the number of packets lost (and not the number of audio samples that couldn't be regenerated). This site behaved quite badly and shows a lot of losses (10 to 27 percent).



Graph 32

Graph 32 above shows the loss fraction for a domestic site in Brandon, Florida, both while transmitting the continuos audio stream and the discontinuous audio stream. 2-way interleaving was used. This is the total loss fraction, i.e. the number of packets lost (and not the number of audio samples that couldn't be regenerated). Apart from one big spike, the losses are consistently quite low.

8 Conclusions

When using UDP echo, the losses were a little higher on the international sites than on the domestic one. Students.uiuc.edu is an exception. Hence, the average SNR at the domestic site was higher than the one at international sites.

As explained earlier, we tested the efficacy of linear and quadratic smoothing using two different kind of audio samples. Linear interpolation was found to be consistently better than quadratic interpolation on the audio file with discontinuous audio (a voice counting numbers with pauses). This is because linear interpolation fits piecewise linear curves to the missing samples. On the other hand, quadratic interpolation outperformed linear on the audio file with smoothly varying signal. This is because quadratic interpolation tries to fit a smoother, quadratic curve to the missing samples.

SNR also shows a direct relation to the amount of loss—it is lower for higher loss values, just as expected.

International sites have greater average jitters and thus the estimates of jitter buffer size are higher for such sites (nearly 20). However the number of instances when the number of packets in the buffer goes to zero, which is an indicator of how effectively the traffic is being smoothed, varies from site to site. Thus while smoothing is effective across all sites, traffic from some sites, those that are lightly loaded, can be more effectively smoothed than traffic from other, more congested, sites.

Smoothing of jitters is equally efficient in the case of TCP encapsulated in UDP. Other behavior is also completely similar except for the fact that jitter of packets received from the network are slightly higher in this case possibly due to the bursty non-rate-controlled behavior of UDP. Since the transmission from the local machine to the remote one is over UDP, we see more losses and thus more retransmission. Thus the average jitter is also higher. Consequently the jitter buffer sizes needed for this phase were also roughly 1.5 to 2 times that of the former.

Results present in this paper could not be directly compared with those present in [2] as most of the results presented in the latter involved different kinds of experimentation (different interleaving factors for the same hosts, transformation matrices) etc. However, the semantics and the conclusions of this lab agree with the general picture presented in [2]. The values of jitter, end-to-end delays and SNR's observed in Fig 10 of [2] are very similar to those observed in this report.

References

[1] Sam Liang and David Cheriton *TCP-RTM: Using TCP for Real-Time applications*

[2] Benjamin Wah and Dong Lin *Transformation-Based Reconstruction for Real-Time Voice Transmissions over the Internet*

[3] Quadratic Interpolation *http://mpec.sc.mahidol.ac.th/numer/STEP21.HTM*

A TCP Echo, Application Code

A.1 sender.cc

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <linux/socket.h>

//include files for threads
#include <pthread.h>
#include <semaphore.h>

extern int errno;

//global socket identifiers
int s, recv_s;

//global var for threads
struct timespec rate;
pthread_cond_t cond, cond1, cond2;
pthread_mutex_t mutex, dummy, dummy2;

//some of the above variables are used for rate control while others
//are used for synchronization

//structure to store each buffer entry
struct bufItem{
  //deletion at the head, so only need next
  struct bufItem *next;
  //the data of the packet
  char buf[COMP_SIZE];
};
typedef struct bufItem BufItem;

//buffer
BufItem *head, *tail;

int numPackets, jitterBufSize, start;
```

```
bool record(AUDIOFMT* audioBuf)
{
    int readNum;

    readNum=read(audio_fd, (char*)audioBuf, BUF_SIZE_BYTE);

    if (readNum!=BUF_SIZE_BYTE)
        return false;

    return true;
}

int pack_output(unsigned code,int bits, unsigned char* encodeBuf)
{
    static unsigned int out_buffer = 0;
    static int      out_bits = 0;
    unsigned char   out_byte;

    static int i=0;

    out_buffer |= (code << out_bits);
    out_bits += bits;
    if (out_bits >= 8)
    {
        out_byte = out_buffer & 0xff;
        out_bits -= 8;
        out_buffer >>= 8;
        encodeBuf[i]=out_byte;
        i=(i+1)%COMP_SIZE;
    }
    return (out_bits > 0);
}


void encode(AUDIOFMT* audioBuf, unsigned char* encodeBuf)
{
    unsigned char       code;
    int         resid;

    int i;
```

```
    static int encodeFirst=1;
    static struct g72x_state  enstate;

    if (encodeFirst==1)
    {
// This segment should be called for the first frame only.
// Subsequent frames should bypass this part of the loop.
        g72x_init_state(&enstate);
        encodeFirst=0;
    }

    for (i=0;i<BUF_SIZE;i++)
    {
        code=g721_encoder(audioBuf[i],AUDIO_ENCODING_ULAW, &enstate);
        resid=pack_output(code,4, encodeBuf);
    }
    while (resid)
    {
        assert(false);
        resid = pack_output(0, 4, encodeBuf);
    }

}

void *forward(void *ptr){
  //this thread consumes packets periodically from the buffer and
  //forwards them to a receiver

  struct timeval tm;
  char sbuf[COMP_SIZE];

  while(1){
    //pass it out to the local receiver

    //lock the buffer
    if(pthread_mutex_lock(&mutex) != 0){
      perror("pthread_mutex_lock");
      exit(3);
    }
```

```
    if(numPackets == 0){
      //if nothing in the buffer, wait till buffer fills up to the
      //jitterBufSize
      start=1;

      printf("Going to wait.\n");

      if(pthread_cond_wait(&cond1, &mutex)!=0){
perror("pthread_cond_wait");
exit(3);
      }

      printf("Done waiting.\n");
    }


    //We have "enough" packets in the buffer, start sending

    //dequeue
    BufItem *b = head;
    head = head->next;
    numPackets--;

    //release lock
    if(pthread_mutex_unlock(&mutex) != 0){
      perror("pthread_mutex_unlock");
      exit(3);
    }

    bcopy(b->buf, sbuf, COMP_SIZE);

    //forward it to sender
    if(send(recv_s, sbuf, COMP_SIZE, 0) < COMP_SIZE){
      perror("send to local: ");
      exit(3);
    }

    //go to sleep --> simulates periodic consumption by a multimedia
    //application
```

```
    int a1, a2=0, a3;
    a1=pthread_mutex_lock(&dummy);
gettimeofday(&tm, NULL);

double abc=tm.tv_sec + (tm.tv_usec)/1000000.0;
abc+=0.01;
rate.tv_sec = (int) abc;
rate.tv_nsec= (int)((abc - rate.tv_sec)*1000000000);

while (a2!=ETIMEDOUT)
  a2=pthread_cond_timedwait(&cond, &dummy, &rate); //simulate constant writing rate
    a3=pthread_mutex_unlock(&dummy);
  }

}

void *recv_network(void *ptr){
  //no input...

  //receives data from the network, (Echoed) and writes
  //into the buffer (Producer)

  char tbuf[COMP_SIZE];
  int i=0;
  int flag;
  struct timeval tm;
  int sec=0, usec=0;

  double jitter=0.0;
  double delay=0.0;

  while(1){

    int datalen, fraglen;
    flag=1;

    //receive a pkt of size COMP_SIZE
    for(datalen=0, fraglen=0; datalen<COMP_SIZE; ){
      if((fraglen=recv(s, tbuf+datalen, COMP_SIZE - datalen, 0)) == -1){
```

```
perror("recv from remote: ");
exit(2);
      }
      datalen+=fraglen;
    }

    //create a bufItem
    BufItem *n = (BufItem *)malloc(sizeof(BufItem));
    bcopy((const void *)tbuf, (void *)n->buf, COMP_SIZE);
    n->next = NULL;

    //store it into the buffer
    //1. acquire a lock
    if(pthread_mutex_lock(&mutex) != 0){
      perror("pthread_mutex_lock");
      exit(3);
    }

    gettimeofday(&tm, NULL);

    //2. compute jitter, delay etc and update jitterBufSize
    //the calculation has been explained in detail in the report
    if(sec!=0) {

      double curr_delay = (tm.tv_sec -sec) + (tm.tv_usec -usec)/1000000.0;
      if(delay !=0){
double dd = (curr_delay - delay);
if(dd < 0)
  dd = -1*(dd);

jitter = ((i-1)*jitter + dd)/(i);
jitterBufSize = (int)((jitter/0.01) * 4);

if(jitterBufSize < 3) jitterBufSize=3;
      }

      delay = ((i)*delay + curr_delay)/(i+1);
    }
    sec=tm.tv_sec;
    usec=tm.tv_usec;
```

```
    //update buffer
    if(tail != NULL) tail->next=n;
    tail=n;
    if(head==NULL) head = n;

    numPackets++;

    //if forwarder is blocked (due to empty buffer), wake it up when
    //there are atleast jitterBufSize packets in the buffer
    if(start==1 && numPackets>=jitterBufSize){
      start=0;
      flag=0;

      //Waking up consumer
      if(pthread_mutex_unlock(&mutex) != 0){
perror("pthread_mutex_unlock");
exit(4);
      }

      if(pthread_cond_signal(&cond1)!=0){
perror("pthread_cond_signal");
exit(6);
      }
    }

    //release lock
    if(flag==1 && pthread_mutex_unlock(&mutex) != 0){
      perror("pthread_mutex_unlock");
      exit(4);
    }

  }
}

void main(int argc, char *argv[])
{
    AUDIOFMT audioBuf[BUF_SIZE];
    unsigned char encodeBuf[COMP_SIZE];
```

```
FILE* fpen;
FILE* fpen1;

//two threads apart from the parent
pthread_t recver, fwder;

struct timeval tm;

//recorded stream and the codedstream are printed out
fpen=fopen("./codedstream", "wb");
fpen1 = fopen("./orgstream", "wb");

assert(fpen!=NULL);
assert(fpen1!=NULL);

//create the socket
struct hostent *he;
int datalen;
struct sockaddr_in rs;
struct sockaddr_in ls;

//s, used to send to TCP Echo server
if((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
  perror("socket creation failed");
  exit(1);
}

//recv_s used to forward the captured stream
if((recv_s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
  perror("socket creation failed");
  exit(1);
}

bzero(&rs, sizeof(struct sockaddr_in));
bzero(&ls, sizeof(struct sockaddr_in));

if(argc<1){
  printf("Command line argument for remote tcp echo server's IP is needed\n");
  exit(3);
}
```

```
// get the address of the remote tcp echo server from the name/IP
// specified on the command line
if((he = gethostbyname(argv[1])) == NULL) {
  perror("gethostbyname failed");
  exit(1);
}

// fill in the information about the server address in the sockaddr_in
rs.sin_family = AF_INET;
rs.sin_port =  htons(7);
rs.sin_addr = *((struct in_addr *)(he->h_addr));

//address to which the stream has to be forwarded
ls.sin_family = AF_INET;
ls.sin_port = htons(TCP_RPORT);
inet_aton(RADDR, (struct in_addr *)&ls.sin_addr);

//connect to the local receiver
if(connect(recv_s, (struct sockaddr *)&ls, sizeof(struct sockaddr_in)) == -1) {
  perror("connect recv failed: ");
  exit(1);
}

//try connecting to the remote TCP Echo server
if(connect(s, (struct sockaddr *)&rs, sizeof(struct sockaddr_in)) == -1) {
  perror("connect echo failed: ");
  exit(1);
}

//initialize synchronization variables before spawning threads
pthread_cond_init(&cond, NULL);
pthread_cond_init(&cond1, NULL);
pthread_cond_init(&cond2, NULL);
pthread_mutex_init(&mutex, NULL);
pthread_mutex_init(&dummy, NULL);
pthread_mutex_init(&dummy2, NULL);

//other shared variables
head=NULL;
```

```
    tail=NULL;
    numPackets=0;
    jitterBufSize=10;
    start=1;

    //spawn receiver out
    if(pthread_create(&recver, NULL, &recv_network, (void *)NULL) == -1){
      perror("Pthread_create");
      exit(3);
    }

    //spawn forwarded out
    if(pthread_create(&fwder, NULL, &forward, (void *) NULL) == -1){
      perror("pthread_create");
      exit(4);
    }

    //parent does the sending job...
    open_audio();

    while(1)
    {
        if (!record(audioBuf))
            break;

        encode(audioBuf, encodeBuf);

        //coded stream in encodeBuf
        //sending out to remote

if(send(s, encodeBuf, COMP_SIZE, 0) < COMP_SIZE){
  perror("send");
  exit(5);
}

//storing streams for later use
        fwrite(encodeBuf, sizeof(char), COMP_SIZE, fpen);
        fflush(fpen);
fwrite(audioBuf, sizeof(char), BUF_SIZE, fpen1);
fflush(fpen1);
```

```
//sleep for some time, used as a rate throttle
//audio sample of 1k takes more than 30mSec to be generated
//since each pkt is 0.5k we sleep for about 10msec
//this is less than the required

int a1, a2=0, a3;
a1=pthread_mutex_lock(&dummy2);

gettimeofday(&tm, NULL);

double abc=tm.tv_sec + (tm.tv_usec)/1000000.0;
abc+=0.01;
rate.tv_sec = (int) abc;
rate.tv_nsec= (int)((abc - rate.tv_sec)*1000000000);

while(a2 != ETIMEDOUT)
  a2=pthread_cond_timedwait(&cond2, &dummy2, &rate);

a3=pthread_mutex_unlock(&dummy2);
    }

    fclose(fpen);
    fclose(fpen1);

    close_audio();
}
```

A.2 recver.cc

```
#include "public.h"
#include "common.h"
#include "g72x.h"
#include "audio.h"

// include files for socket communication
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <cstring>
#include <errno.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <linux/socket.h>

bool play(AUDIOFMT* decodeBuf)
{
    if (write(audio_fd, (char*)&decodeBuf[0], BUF_SIZE_BYTE)!= BUF_SIZE_BYTE)
        return false;

    return true;
}

int unpack_input(unsigned char *code, int bits, unsigned char* encodeBuf)
{
    static unsigned int in_buffer = 0;
    static int      in_bits = 0;
    unsigned char       in_byte;

    static int i=0;

    if (in_bits < bits)
    {
        in_byte=encodeBuf[i];
```

```
        i=(i+1)%COMP_SIZE;
        in_buffer |= (in_byte << in_bits);
        in_bits += 8;
    }
    *code = in_buffer & ((1 << bits) - 1);
    in_buffer >>= bits;
    in_bits -= bits;
    return (in_bits > 0);
}


void decode(AUDIOFMT* decodeBuf, unsigned char* encodeBuf)
{
    unsigned char       code;
    signed short sample;

    int i;

    static int decodeFirst=1;
    static struct g72x_state  destate;

    if (decodeFirst==1)
    {
  // This segment should be called for the first frame only.
// Subsequent frames should bypass this part of the loop.
        g72x_init_state(&destate);
        decodeFirst=0;
    }

    for (i=0;i<BUF_SIZE;i++)
    {
        unpack_input(&code,4, encodeBuf);
        //decodeBuf[i]=(short)g721_decoder(code,AUDIO_ENCODING_LINEAR,&destate);
        decodeBuf[i]=(short)g721_decoder(code,AUDIO_ENCODING_ULAW,&destate);
    }

}

void main(int argc, char *argv[])
{
```

```
AUDIOFMT decodeBuf[BUF_SIZE];
unsigned char encodeBuf[COMP_SIZE];

FILE *fpen1;

//write the decoded stream
fpen1 = fopen("./decstream", "wb");
assert(fpen1!=NULL);

//initialize the socket to hear on...
struct hostent *he;
int s, conn_s;
int datalen;
struct sockaddr_in ls;

//s, recv from local forwarder
if((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
  perror("socket creation failed");
  exit(1);
}

bzero(&ls, sizeof(struct sockaddr_in));

ls.sin_family = AF_INET;
ls.sin_port = htons(TCP_RPORT);
inet_aton(RADDR, (struct in_addr *)&ls.sin_addr);

//bind s, such that the remote forwarder can connect to me
if(bind(s, (struct sockaddr *)&ls, sizeof(struct sockaddr_in)) == -1) {
  perror("bind recv failed: ");
  exit(1);
}

//listen and accept a connect
if(listen(s, 5) == -1){
  perror("listen recv: ");
  exit(2);
}

int ln=sizeof(struct sockaddr_in);
```

```
    if((conn_s=accept(s, (struct sockaddr *)&ls, (socklen_t *)&ln)) == -1){
      perror("accept recv: ");
      exit(3);
    }

    //receiving data from conn_s and playing it out

    open_audio();

    //used to estimate jitter at this end
    int i=0;
    struct timeval tm;
    int sec=0, usec=0;
    double jitter=0.0;
    double delay=0.0;

    while(1)
    {
      //to get one full packet off the stream
      for(int datalen=0, fraglen=0; datalen<COMP_SIZE; ){
if((fraglen=recv(conn_s, encodeBuf+datalen, COMP_SIZE - datalen, 0)) == -1){
  perror("recv from forwarded: ");
  exit(4);
}
datalen+=fraglen;
      }

      decode(decodeBuf, encodeBuf);

      //estimate jitter at this end (after smoothing)
      gettimeofday(&tm, NULL);
      if(sec!=0){
double curr_delay = (tm.tv_sec -sec) + (tm.tv_usec -usec)/1000000.0;

if(delay !=0){
  double dd = (curr_delay - delay);
  if(dd < 0)
    dd = -1*(dd);

  jitter = (i*jitter + dd)/(i+1);
```

```
  printf("\t Jitter: %f\n", jitter);
}

delay = ((i)*delay + curr_delay)/(i+1);

      }
      sec=tm.tv_sec; usec=tm.tv_usec;

      //store the decoded stream
      fwrite(decodeBuf, sizeof(char), BUF_SIZE, fpen1);
      fflush(fpen1);

      if (!play(decodeBuf))
break;
    }

    fclose(fpen);
    fclose(fpen1);

    close_audio();
}
```

A.3 common.h

```
#ifndef COMMON_H
#define COMMON_H

typedef unsigned char AUDIOFMT;

//size of each packet
#define BUF_SIZE 512

//Port and address of the forwarder
#define TCP_RPORT 8786
#define RADDR "192.168.0.15"

/* if AUDIOFMT is unsigned char */
#define BUF_SIZE_BYTE (BUF_SIZE)
#define COMP_SIZE (BUF_SIZE_BYTE/2)
#define PKT_SIZE (BUF_SIZE_BYTE/2)

#define INTLV_DEGREE 2
#define INTLV_BUF_SIZE (BUF_SIZE_BYTE/INTLV_DEGREE)
#define INTLV_COMP_SIZE (COMP_SIZE/INTLV_DEGREE)

typedef struct pkt Pkt;
#endif
```

B UDP Echo, Application code

B.1 sender.cc

```
#include "public.h"
#include "common.h"
#include "g72x.h"
#include "audio.h"

// include files for socket communication
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <cstring>
#include <errno.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <linux/socket.h>

//include files for threads
#include <pthread.h>
#include <semaphore.h>

extern int errno;

//global socket identifiers
int s, recv_s;

//global vars for threads
pthread_cond_t cond;
pthread_mutex_t mutex;

FILE  *fpen, *fpen1;

bool record(AUDIOFMT* audioBuf)
```

```
{
  //...snipped
}

int pack_output(unsigned code,int bits, unsigned char* encodeBuf)
{
  //...snipped
    if (out_bits >= 8)
    {
        out_byte = out_buffer & 0xff;
        out_bits -= 8;
        out_buffer >>= 8;
        encodeBuf[i]=out_byte;
//change
        i=(i+1)%INTLV_COMP_SIZE;
    }
    return (out_bits > 0);
}


void encode(AUDIOFMT* audioBuf, unsigned char* encodeBuf)
{

  //...snipped
  for (i=0;i<INTLV_BUF_SIZE;i++){
    //code=g721_encoder(audioBuf[i],AUDIO_ENCODING_LINEAR, &enstate);
    code=g721_encoder(audioBuf[i],AUDIO_ENCODING_ULAW, &enstate);
    resid=pack_output(code,4, encodeBuf);
  }
  while (resid)
    {
      assert(false);
      resid = pack_output(0, 4, encodeBuf);
    }
}

void *recv_network(void *ptr){
  //receive off the network and forward to the application receiving the data
  unsigned int numLosses=0;
  unsigned int recvid=0; //the next packet to recv
```

```
  int pSize = sizeof(Pkt); //the size of each pkt

  Pkt pkt; //the variable to store the incoming packet


  while(1){
    int datalen, fraglen;

    for(datalen=0, fraglen=0; datalen<pSize; ){
      if((fraglen=recv(s, ((char *)&pkt)+datalen, pSize - datalen, 0)) == -1){
perror("recv from remote: ");
exit(2);
      }
      datalen+=fraglen;
    }
    //got a packet, check id and send it out
    if(pkt.tupleid < recvid) {
      continue; //out-of-order => loss, already handled
    }

    //check the tupleid
    if(pkt.tupleid >= recvid){
      numLosses += (pkt.tupleid - recvid);
      recvid = pkt.tupleid+1;

      //send it out
      if(send(recv_s, &pkt, pSize, 0) < pSize){
perror("send");
exit(4);
      }
    }
  }
}

void main(int argc, char *argv[])
{
    AUDIOFMT audioBuf[BUF_SIZE];
    AUDIOFMT intLvBuf[INTLV_DEGREE][INTLV_BUF_SIZE];
    unsigned char encodeBuf[INTLV_DEGREE][INTLV_COMP_SIZE];
    unsigned int id=0;
```

```
Pkt pckt, pckt1;
struct hostent *he;

unsigned int len;
int datalen;

struct sockaddr_in rs;
struct sockaddr_in ls;

pthread_t recver;

fpen=fopen("./codedstream", "wb");
fpen1 = fopen(argv[2], "rb");

assert(fpen!=NULL);
assert(fpen1!=NULL);

// create a UDP socket to connect to remote UDP Echo server
if((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
  perror("socket creation failed");
  exit(1);
}

// create a UDP socket for connection with the application
if((recv_s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
  perror("recv socket");
  exit(1);
}

bzero(&rs, sizeof(struct sockaddr_in));
bzero(&ls, sizeof(struct sockaddr_in));

if(argc < 1){
  printf("IP of remote echo server needed!\n");
  exit(3);
}

// get the address of the remote udp echo server from the name/IP
// specified on the command line
if((he = gethostbyname(argv[1])) == NULL) {
```

```
    perror("gethostbyname failed");
    exit(1);
  }

  // fill in the information about the server address in the sockaddr_in
  rs.sin_family = AF_INET;
  rs.sin_port =  htons(7);
  rs.sin_addr = *((struct in_addr *)(he->h_addr));

  //fill in address of application
  ls.sin_family = AF_INET;
  ls.sin_port = htons(atoi(argv[3]));
  inet_aton(RADDR, (struct in_addr *)&ls.sin_addr);


  if(connect(recv_s, (struct sockaddr *)&ls, sizeof(struct sockaddr_in)) == -1) {
    perror("connect to receiver");
    exit(1);
  }


  // try connecting to the remote UDP server
  // saves the trouble of using the sendto and recvfrom with server addree
  // each time :-)

  if(connect(s, (struct sockaddr *)&rs, sizeof(struct sockaddr_in)) == -1) {
    perror("connect");
    exit(1);
  }

  //initialize synchronization variables
  pthread_cond_init(&cond, NULL);
  pthread_mutex_init(&mutex, NULL);

  //some time values
  struct timespec rate;
  struct timeval tm;

  //spawn threads
```

```
    if(pthread_create(&recver, NULL, &recv_network, (void *)NULL) == -1){
      perror("Pthread_create");
      exit(3);
    }

    //the parent records and sends out the audio
    open_audio();

    while(1)
    {

      if (!record(audioBuf))
break;

      /* Interleave the audioBuf */
      int ptr=0;
      while(ptr < BUF_SIZE)
for(int i=0; i<INTLV_DEGREE; i++, ptr++)
  intLvBuf[i][ptr/INTLV_DEGREE] = audioBuf[ptr];

      /* Encode each stream */
      for(int i=0;i<INTLV_DEGREE;i++)
encode(intLvBuf[i], encodeBuf[i]);

      //coded streams in encodeBuf

      /* send and receive all the streams */
      for(int i=0;i<INTLV_DEGREE;i++) {
//construct a packet
bcopy(encodeBuf[i], pckt.data, INTLV_COMP_SIZE);
pckt.tupleid = id++;

if (send(s, &pckt, sizeof(Pkt), 0) == -1) {
  perror("sendto");
  exit(1);
}

fwrite(&pckt, sizeof(char), sizeof(Pkt), fpen);
fflush(fpen);
```

```
// pause for some time
pthread_mutex_lock(&mutex);
gettimeofday(&tm, NULL);

double abc=tm.tv_sec + (tm.tv_usec)/1000000.0;
abc+=0.05/INTLV_DEGREE;
rate.tv_sec = (int) abc;
rate.tv_nsec= (int)((abc - rate.tv_sec)*1000000000);

while (pthread_cond_timedwait(&cond, &mutex, &rate) != ETIMEDOUT);
//simulate constant writing rate
pthread_mutex_unlock(&mutex);
      }

      fwrite(audioBuf, sizeof(char), BUF_SIZE, fpen1);
      fflush(fpen1);
    }

    fclose(fpen);
    fclose(fpen1);

    close_audio();
}
```

B.2 recver.cc

```
//...include files snipped

bool play(AUDIOFMT* decodeBuf)
{
  //...snipped
}

int unpack_input(unsigned char *code, int bits, unsigned char* encodeBuf)
{
  //...snipped
  if (in_bits < bits)
    {
      in_byte=encodeBuf[i];
```

```
      i=(i+1)%INTLV_COMP_SIZE;
        in_buffer |= (in_byte << in_bits);
        in_bits += 8;
    }
  *code = in_buffer & ((1 << bits) - 1);
  in_buffer >>= bits;
  in_bits -= bits;
  return (in_bits > 0);
}


void decode(AUDIOFMT* decodeBuf, unsigned char* encodeBuf)
{

  //...snipped
  for (i=0;i<INTLV_BUF_SIZE;i++)
    {
      unpack_input(&code,4, encodeBuf);
      //decodeBuf[i]=(short)g721_decoder(code,AUDIO_ENCODING_LINEAR,&destate);
      decodeBuf[i]=(short)g721_decoder(code,AUDIO_ENCODING_ULAW,&destate);
    }
}

void main(int argc, char *argv[])
{
    AUDIOFMT decodeBuf[BUF_SIZE];
    unsigned char encodeBuf[COMP_SIZE];
    unsigned char intlvDecodeBuf[INTLV_DEGREE][INTLV_BUF_SIZE];
    bool recvd[INTLV_DEGREE];
    unsigned char buffer[sizeof(Pkt)];
    unsigned char accum[sizeof(Pkt)];
    Pkt pckt, pckt1;
    FILE* fpen;
    FILE *fpen1;
    unsigned int len;
    int curr_tuple=0;
    int tupid, streamid;
    int datalen, ret;
    int s;
    struct sockaddr_in rs;
```

```
    struct sockaddr_in ls;

    fpen1 = fopen("./decstream", "wb");
    assert(fpen1!=NULL);

    open_audio();

    // initialize the recvd bit vector
    for(int i=0;i<INTLV_DEGREE;i++)
      recvd[i] = false;

    /* create a server (listening) socket */
    if((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
      perror("listening socket");
      exit(1);
    }

    bzero(&rs, sizeof(struct sockaddr_in));
    bzero(&ls, sizeof(struct sockaddr_in));

    ls.sin_family = AF_INET;
    ls.sin_port = htons(atoi(argv[1]));
    inet_aton(RADDR, (struct in_addr *)&ls.sin_addr);

    //bind, listen and accept a connection from the forwarder
    if(bind(s, (struct sockaddr *)&ls, sizeof(struct sockaddr_in)) == -1) {
      perror("bind");
      exit(1);
    }

    while(1)
    {
      /* read the stream from the network */

      datalen = sizeof(Pkt);
      while(datalen > 0) {
if((ret = recv(s, buffer, datalen, 0)) == -1) {
  perror("recv");
  exit(1);
}
```

```
bcopy(buffer, &accum[sizeof(Pkt) - datalen], ret);
datalen -= ret;
      }

      // one packet complete
      bcopy(accum, &pckt, sizeof(Pkt));

      tupid = (pckt.tupleid)/INTLV_DEGREE;
      streamid = (pckt.tupleid)%INTLV_DEGREE;

      if(tupid == curr_tuple) {
//the packet belongs to the currently stored tuple
//blindly store the packet
recvd[streamid] = true;
decode(intlvDecodeBuf[streamid], pckt.data);
      }
      else {
// either the current tuple is full or something went amiss
// reconstruct the current tuple (if needed) and play it
//the individual codes for liner and quadratic interpolation occur here
//... this code is attached later in this report

//check for and interpolate losses

//.... snipped (occurs later in the report,
// in the following subsections)

// now the entire decode buf is reconstructed, let's play

//de-interleaving
int ptr=0;
while(ptr < BUF_SIZE)
  for(int i=0; i<INTLV_DEGREE; i++, ptr++)
    decodeBuf[ptr] = intlvDecodeBuf[i][ptr/INTLV_DEGREE];

//store the decoded stream
fwrite(decodeBuf, sizeof(char), BUF_SIZE, fpen1);
fflush(fpen1);

//play it
```

```
if (!play(decodeBuf))
  break;

//handle the current packet
curr_tuple++;

while(curr_tuple != tupid) {
  // => Whole tuple is lost!
  //store silence
  bzero(decodeBuf, BUF_SIZE);
  fwrite(decodeBuf, sizeof(char), BUF_SIZE, fpen1);
  fflush(fpen1);

  //play it
  if (!play(decodeBuf))
    break;
  //move ahead
  curr_tuple++;
}
//okay all previous tuples have been handled

//initialize the recv array for the current tuple
for(int i=0;i<INTLV_DEGREE;i++)
  recvd[i] = false;

//set the correct bit in the recv array, corresponding to the current tuple
recvd[streamid] = true;
decode(intlvDecodeBuf[streamid], pckt.data);
      }
    }

    fclose(fpen1);
    close_audio();
}
```

B.3 common.h

```
#ifndef COMMON_H
#define COMMON_H
```

```
typedef unsigned char AUDIOFMT;

//larger packet size
#define BUF_SIZE 1024

/* if AUDIOFMT is unsigned char */
#define BUF_SIZE_BYTE (BUF_SIZE)
#define COMP_SIZE (BUF_SIZE_BYTE/2)

//handling interleaving
#define INTLV_DEGREE 2
#define INTLV_BUF_SIZE (BUF_SIZE_BYTE/INTLV_DEGREE)
#define INTLV_COMP_SIZE (COMP_SIZE/INTLV_DEGREE)

struct pkt {
  unsigned int tupleid;   // the last 1/2 bits serve as the stream identifier
  unsigned char data[INTLV_COMP_SIZE];
};

typedef struct pkt Pkt;

#define RPORT 3450
#define RADDR "192.168.0.15"

#define NUM_PCKTS 5000

#endif
```

B.4 Interpolation

The following is the code that was plugged above in recver.cc to handle interpolation of losses.

B.4.1 Linear Interpolation

```
int x, y;
// using linear interpolation
```

```
for(int i=0; i<INTLV_DEGREE; i++) {
  //reconstruct each lost sample
  if(recvd[i] == false) {
    // a missing sample
    // try to find the earlier adjacent sample
    int j,k,l;
    bool flag = false;
    x = y = 0;
    for(j=1; j<INTLV_DEGREE;j++) {
      l = i-j;
      if(l<0) {
x = -1; //previous tuple
l = INTLV_DEGREE + l; //tuple id of the earlier non-zero sample
      }
      if(recvd[l]) {
flag = true;
break;
      }
    }
    if(!flag) {
      // no packets in stream!
      // feed silence into everything
      for(j=0; j<INTLV_DEGREE; j++)
bzero(intlvDecodeBuf[j], INTLV_BUF_SIZE);
      break;
    }
    //else found an earlier packet, go ahead and find the
    //later one...
    j = l;
    for(k=1; k<INTLV_DEGREE; k++) {
      l = i + k;
      if(l>=INTLV_DEGREE) {
y = 1; //the next tuple
l = l - INTLV_DEGREE; //the id of the later sample
      }
      if(recvd[l])
break;
    }
    k=l;
```

```
    // now reconstruct the missing parts
    int lhs, rhs, j1, k1;
    for(int m=0; m < INTLV_BUF_SIZE; m++) {
      if(m+x < 0) {
intlvDecodeBuf[i][m] = intlvDecodeBuf[k][m]; // y wud be zero
continue;
      }
      else lhs = m+x;
      if(m+y >= INTLV_BUF_SIZE) {
intlvDecodeBuf[i][m] = intlvDecodeBuf[j][m]; // x wud be zero
continue;
      }
      else rhs = m+y;

      intlvDecodeBuf[i][m] = (intlvDecodeBuf[j][lhs] + intlvDecodeBuf[k][rhs])/2;
    }
  }
}
```

B.4.2 Quadratic Interpolation

```
int x, y1, y2; //three samples have to be found!!

// using quadratic interpolation
for(int i=0; i<INTLV_DEGREE; i++) {
  //reconstruct each lost tuple
  if(recvd[i] == false) {
    printf("\t some lost tuples.\n");
    // a missing sample
    // try to find the earlier adjacent sample
    int j,k,k1,k2,l;
    bool flag = false;
    x = y1 = y2  = 0;
    for(j=1; j<INTLV_DEGREE;j++) {
      l = i-j;
      if(l<0) {
x = -1;
l = INTLV_DEGREE + l;
      }
```

```
      if(recvd[l]) {
flag = true;
break;
      }
    }
    if(!flag) {
      // no packets in stream!
      // feed silence into everything
      for(j=0; j<INTLV_DEGREE; j++)
bzero(intlvDecodeBuf[j], INTLV_BUF_SIZE);
      break;
    }
    //found one sample earlier... to find two ahead of me
    j = l;
    flag=false;
    for(k=1; k<2*INTLV_DEGREE; k++) {
      l = i + k;
      if(!flag && l>=INTLV_DEGREE) {
y1 = 1;
l = l - INTLV_DEGREE;
      }
      if(flag && l>=INTLV_DEGREE && l<2*INTLV_DEGREE) {
//both the non-zero samples are in the tuple ahead
y2=1;
l = l - INTLV_DEGREE;
      }
      if(flag && l>=2*INTLV_DEGREE) {
//the second sample ahead of me is in the second tuple from me
y2=2;
l = l - (2*INTLV_DEGREE);
      }
      if(recvd[l] && flag)
break;
      if(recvd[l]) {
k1 = l;
flag = true;
      }
    }
    k2=l;
    // now reconstruct the missing parts
```

```
    int lhs, rhs, rrhs;
    for(int m=0; m < INTLV_BUF_SIZE; m++) {
      if(m+x < 0) {
intlvDecodeBuf[i][m] = intlvDecodeBuf[k1][m]; // y wud be zero
continue;
      }
      else lhs = m+x;
      if(m+y1 >= INTLV_BUF_SIZE) {
intlvDecodeBuf[i][m] = intlvDecodeBuf[j][m]; // x wud be zero
continue;
      }
      else {
rhs = m + y1;
if(m+y2 >= INTLV_BUF_SIZE) {
  intlvDecodeBuf[i][m] = (intlvDecodeBuf[j][lhs] + intlvDecodeBuf[k1][rhs])/2;
  continue;
}
else
  rrhs = m + y2;
      }
      intlvDecodeBuf[i][m] = intlvDecodeBuf[j][lhs] + (intlvDecodeBuf[k1][rhs] - intlvDecodeBuf[j][lhs])/2 - (intlvDecodeBuf[j][lhs] - 2*intlvDecodeBuf[k1][rhs] + intlvDecodeBuf[k2][rrhs])/8;
    }
  }
}
```

C TCP over UDP, Application code

C.1 sender.cc

```
//...include files snipped

extern int errno;

//global socket identifiers
int s;

bool record(AUDIOFMT* audioBuf)
{
  //...snipped
}

int pack_output(unsigned code,int bits, unsigned char* encodeBuf)
{
  //...snipped
}


void encode(AUDIOFMT* audioBuf, unsigned char* encodeBuf)
{
  //...snipped
}

void main(int argc, char *argv[])
{
    AUDIOFMT audioBuf[BUF_SIZE];
    unsigned char encodeBuf[COMP_SIZE];

    FILE* fpen;
    FILE* fpen1, *freader;

    struct timeval tm;

    fpen=fopen("./codedstream", "wb");
    fpen1 = fopen("./orgstream", "wb");
```

```
assert(fpen!=NULL);
assert(fpen1!=NULL);

//create the socket
struct hostent *he;
int datalen;
struct sockaddr_in rs;

//initialize the TCP socket to send to the remote TCP Echo port
if((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
  perror("socket creation failed");
  exit(1);
}

bzero(&rs, sizeof(struct sockaddr_in));

if(argc<1){
  printf("Command line argument for remote tcp echo server needed\n");
  exit(3);
}

// get the address of the remote tcp echo server from the name/IP
// specified on the command line
if((he = gethostbyname(argv[1])) == NULL) {
  perror("gethostbyname failed");
  exit(1);
}

// fill in the information about the server address in the sockaddr_in
rs.sin_family = AF_INET;
rs.sin_port =  htons(LRPORT);
inet_aton(LADDR, (struct in_addr*)&rs.sin_addr);


//setsockopt for the remote server as argv[1]
// set TCP_ECHO option
if(setsockopt(s, 6, 13, he->h_addr, he->h_length) == -1) {
  perror("setsockopt failed");
  exit(1);
```

```
    }

    // set maximum segment size
    int maxseg=ECHO_MSS;
    if(setsockopt(s, 6, 2, (char *)&maxseg, sizeof(unsigned long)) == -1) {
      perror("setsockopt failed");
      exit(1);
    }

    //connect to the local receiver
    if(connect(s, (struct sockaddr *)&rs, sizeof(struct sockaddr_in)) == -1) {
      perror("connect local failed: ");
      exit(1);
    }

    //single thread that does the sending job
    open_audio();

    int num=0;
    while(1)
    {
      char tbuf[2];

      if (!record(audioBuf))
break;

      encode(audioBuf, encodeBuf);

      //coded stream in encodeBuf
      //sending out to remote

      if(send(s, encodeBuf, COMP_SIZE, 0) < COMP_SIZE){
perror("send");
exit(5);
      }

      num++;
      if(num%100 == 0)
printf("Sent %d packets.\n", num);
```

```
      //write into buffers
      fwrite(encodeBuf, sizeof(char), COMP_SIZE, fpen);
      fflush(fpen);
      fwrite(audioBuf, sizeof(char), BUF_SIZE, fpen1);
      fflush(fpen1);

      //sleep for some time, used as a rate throttle
      //sample of 1k takes about 30mSec to be generated
      usleep(100); //sleeping 10mSec
    }

    fclose(fpen);
    fclose(fpen1);

    close_audio();
}
```

C.2 recver.cc

```
//this file reads from the network and forwards to the application
//...includes have been snipped

bool play(AUDIOFMT* decodeBuf)
{
  //..snipped
}

int unpack_input(unsigned char *code, int bits, unsigned char* encodeBuf)
{
  //..snipped
}


void decode(AUDIOFMT* decodeBuf, unsigned char* encodeBuf)
{
  //..snipped
}

void main(int argc, char *argv[])
```

```
{
    AUDIOFMT decodeBuf[BUF_SIZE];
    unsigned char encodeBuf[COMP_SIZE];

    FILE *fpen1;

    //write the decoded stream
    fpen1 = fopen("./decstream", "wb");
    assert(fpen1!=NULL);

    //initialize the socket to hear on...
    struct hostent *he;
    int s, conn_s;
    int datalen;
    struct sockaddr_in ls;

    //socket "s", receive from forwarder
    if((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket creation failed");
      exit(1);
    }

    bzero(&ls, sizeof(struct sockaddr_in));

    ls.sin_family = AF_INET;
    ls.sin_port = htons(TCP_RPORT);
    inet_aton(RADDR, (struct in_addr *)&ls.sin_addr);

    //bind s, such that the remote forwarder can connect to me
    if(bind(s, (struct sockaddr *)&ls, sizeof(struct sockaddr_in)) == -1) {
      perror("bind recv failed: ");
      exit(1);
    }

    //listen and accept a connect
    if(listen(s, 5) == -1){
      perror("listen recv: ");
      exit(2);
    }
```

```
    int ln=sizeof(struct sockaddr_in);
    if((conn_s=accept(s, (struct sockaddr *)&ls, (socklen_t *)&ln)) == -1){
      perror("accept recv: ");
      exit(3);
    }

    //receiving data from conn_s and playing it out

    open_audio();

    //used to estimate jitter at this end
    int i=0;
    struct timeval tm;
    int sec=0, usec=0;
    double jitter=0.0;
    double delay=0.0;

    while(1)
    {
      //to get one full packet off the stream
      for(int datalen=0, fraglen=0; datalen<COMP_SIZE; ){
//blocking recv...
if((fraglen=recv(conn_s, encodeBuf+datalen, COMP_SIZE - datalen, 0)) == -1){
  perror("recv from forwarded: ");
  exit(4);
}
datalen+=fraglen;
      }

      decode(decodeBuf, encodeBuf);

      //estimate jitter at this end (after smoothing)
      gettimeofday(&tm, NULL);
      if(sec!=0){
double curr_delay = (tm.tv_sec -sec) + (tm.tv_usec -usec)/1000000.0;

if(delay !=0){
  double dd = (curr_delay - delay);
  if(dd < 0)
    dd = -1*(dd);
```

```
  jitter = (i*jitter + dd)/(i+1);
  printf("\t Jitter: %f\n", jitter);
}

delay = ((i)*delay + curr_delay)/(i+1);

      }
      sec=tm.tv_sec; usec=tm.tv_usec;

      //store the decoded stream
      fwrite(decodeBuf, sizeof(char), BUF_SIZE, fpen1);
      fflush(fpen1);

      if (!play(decodeBuf))
break;
    }

    fclose(fpen);
    fclose(fpen1);

    close_audio();
}
```

C.3 lrecver.cc

```
//includes have been snipped

//this is the application which serves as the
//recipient for the TCP-UPD Echo-TCP data

//data received is stored in a buffer to smooth jitters
//and forwarded to the application (recver.cc)

extern int errno;

//global socket identifiers
int s, recv_s, conn_s;
```

```
//global var for threads
struct timespec rate;
pthread_cond_t cond, cond1;
pthread_mutex_t mutex, mutex1;

//cond and dummy are used for rate control

//structure to store the buffer
struct bufItem{
  struct bufItem *next;
  char buf[COMP_SIZE];
};
typedef struct bufItem BufItem;

//buffer
BufItem *head, *tail;
int numPackets, jitterBufSize, start;

void *forward(void *ptr){
  //the Consumer of the queue
  struct timeval tm;
  char sbuf[COMP_SIZE];

  while(1){
    //forward data

    //lock
    if(pthread_mutex_lock(&mutex) != 0){
      perror("pthread_mutex_lock");
      exit(3);
    }

    if(numPackets == 0){
      //if nothing in the buffer, wait till buffer fills up
      //till the jitterBufSize
      start=1;

      if(pthread_cond_wait(&cond, &mutex)!=0){
perror("pthread_cond_wait");
exit(3);
```

```
      }
    }

    //if we have enough packets in the jitter buffer then we can start
    //sending

    BufItem *b = head;
    head = head->next;
    numPackets--;

    if(pthread_mutex_unlock(&mutex) != 0){
      perror("pthread_mutex_unlock");
      exit(3);
    }

    bcopy(b->buf, sbuf, COMP_SIZE);

    //forward the packet
    if(send(recv_s, sbuf, COMP_SIZE, 0) < COMP_SIZE){
      perror("forward send: ");
      exit(3);
    }

    //sleep for some time, simulating a multimedia application
    int a1, a2=0, a3;
    a1=pthread_mutex_lock(&mutex1);
    gettimeofday(&tm, NULL);
    double abc=tm.tv_sec + (tm.tv_usec)/1000000.0;
    abc+=0.01;
    rate.tv_sec = (int) abc;
    rate.tv_nsec= (int)((abc - rate.tv_sec)*1000000000);
    while (a2!=ETIMEDOUT)
      a2=pthread_cond_timedwait(&cond1, &mutex1, &rate); //simulate constant writing rate
    a3=pthread_mutex_unlock(&mutex1);
  }

}

void *recv_network(void *ptr){
  //Thread
```

```
  //receive from the network
  //update jitter, jitterBufSize estimates
  //Producer for the buffer queue
  char tbuf[COMP_SIZE];
  int i=0;
  int flag;
  struct timeval tm;
  int sec=0, usec=0;

  double jitter=0.0;
  double delay=0.0;

  while(1){

    int datalen, fraglen;
    flag=1;

    //receive a pkt of size COMP_SIZE
    for(datalen=0, fraglen=0; datalen<COMP_SIZE; ){
      printf("Trying to receive\n");
      if((fraglen=recv(conn_s, tbuf+datalen, COMP_SIZE - datalen, 0)) == -1){
perror("recv from remote: ");
exit(2);
      }
      datalen+=fraglen;
    }

    BufItem *n = (BufItem *)malloc(sizeof(BufItem));
    bcopy((const void *)tbuf, (void *)n->buf, COMP_SIZE);
    n->next = NULL;

    if(pthread_mutex_lock(&mutex) != 0){
      perror("pthread_mutex_lock");
      exit(3);
    }

    //received a packet... updating jitter values
    gettimeofday(&tm, NULL);
    double dd=0.0, curr_delay=0.0;
    if(sec!=0) {
```

```
      curr_delay = (tm.tv_sec -sec) + (tm.tv_usec -usec)/1000000.0;
      if(delay !=0){
dd = (curr_delay - delay); //current jitter
if(dd < 0)
  dd = -1*(dd); //absolute value

jitter = ((i-1)*jitter + dd)/(i); //average jitter
jitterBufSize = (int)((jitter/0.01) * 4); //estimate the bufSize

if(jitterBufSize < 3) jitterBufSize=3;
      }

      delay = ((i)*delay + curr_delay)/(i+1); //update average delay
    }
    sec=tm.tv_sec;
    usec=tm.tv_usec;

    if(tail != NULL) tail->next=n;
    tail=n;
    if(head==NULL) head = n;

    numPackets++;

    if(start==1 && numPackets>=jitterBufSize){
      start=0;
      flag=0;

      //Waking up consumer, as it would have slept
      //waiting for buffer to fill uptil jitterBufSize
      if(pthread_mutex_unlock(&mutex) != 0){
perror("pthread_mutex_unlock");
exit(4);
      }

      if(pthread_cond_signal(&cond)!=0){
perror("pthread_cond_signal");
exit(6);
      }
    }
```

```
    if(flag==1 && pthread_mutex_unlock(&mutex) != 0){
      perror("pthread_mutex_unlock");
      exit(4);
    }

  }
}

void main(int argc, char *argv[])
{
    pthread_t fwder;

    //create the socket
    struct sockaddr_in rs;
    struct sockaddr_in ls;

    //initialize the TCP socket to recv UDPEcho replayed pkts
    if((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket creation failed");
      exit(1);
    }

    //to forward the data
    if((recv_s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      perror("socket creation failed");
      exit(1);
    }

    bzero(&rs, sizeof(struct sockaddr_in));
    bzero(&ls, sizeof(struct sockaddr_in));

    //to forward the data
    rs.sin_family = AF_INET;
    rs.sin_port =  htons(TCP_RPORT);
    inet_aton(RADDR, (struct in_addr *)&rs.sin_addr);

    //fill in the information about the server address in the sockaddr_in
    ls.sin_family = AF_INET;
    ls.sin_port = htons(LRPORT);
    inet_aton(LADDR, (struct in_addr *)&ls.sin_addr);
```

```
//connect the forwarding socket
if(connect(recv_s, (struct sockaddr *)&rs, sizeof(struct sockaddr_in)) == -1) {
  perror("connect recv failed: ");
  exit(1);
}

//bind, listen and accept the local connection
if(bind(s, (struct sockaddr *)&ls, sizeof(struct sockaddr_in)) == -1){
  perror("bind");
  exit(2);
}

if(listen(s, 5) == -1){
  perror("listen local: ");
  exit(3);
}

int ln = sizeof(struct sockaddr_in);

//accept from local->udpecho
if((conn_s=accept(s, (struct sockaddr *)&rs, (socklen_t *)&ln)) == -1) {
  perror("accept local failed: ");
  exit(1);
}

//read off conn_s and write into recv_s

pthread_cond_init(&cond, NULL);
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex, NULL);
pthread_mutex_init(&mutex1, NULL);

head=NULL;
tail=NULL;
numPackets=0;
jitterBufSize=10;
start=1;

//spawn forwarder
```

```
    if(pthread_create(&fwder, NULL, &forward, (void *) NULL) == -1){
      perror("pthread_create");
      exit(4);
    }

    //parent receives off the network
    recv_network(NULL);
}
```

C.4 common.h

Additional global variables defined as needed.

D Sample data collection scripts: UDP

```
#!/bin/sh

limit=24
ind=0

hosts[0]=students.uiuc.edu
hosts[1]=80.19.88.235
hosts[2]=140.138.138.2
hosts[3]=212.131.108.199
hosts[4]=207.204.210.9

files[0]=/root/lab6/util/orgstream.count
files[1]=/root/lab6/util/orgstream.hmm

#for each iteration, run all the programs
while test "$ind" -lt "$limit";
do
   #run udp experiment for each of the hosts
   date >date."$ind"
   subind=0
   sublimit=${#hosts[*]}

   while test "$subind" -lt "$sublimit";
```

```
   do
       #run for each audio file
       subsubind=0
       subsublimit=${#files[*]}
       while test "$subsubind" -lt "$subsublimit"
       do
          echo Running udp for ${hosts[$subind]}, file ${files[$subsubind]}.

  if test $subind -eq 0 -o $subind -eq 4;
  then
  #domestic site
  ./recver2l >/dev/null &
          ./sender2 "${hosts["$subind"]}" "${files["$subsubind"]}" >/dev/null 2>temp
  #append the lossnum values
  cat temp >> lossnum-2l-"$subsubind"-"$subind"

  #store the signal
  cp decstream decstream-2l-"$subsubind"-"$subind"-"$ind"

  #compute the snr
  ./snr "${files["$subsubind"]} decstream > temp
  cat temp >> snr-2l-"$subsubind"-"$subind"

  #check quadratic interpolation
  ./recver2q >/dev/null &
          ./sender2 "${hosts["$subind"]}" "${files["$subsubind"]}" >/dev/null 2>temp

  #store the new loss
  cat temp >> lossnum-2q-"$subsubind"-"$subind"

  #store the new signal
  cp decstream decstream-2q-"$subsubind"-"$subind"-"$ind"

  #compute and store snr
  ./snr "${files["$subsubind"]} decstream > temp
  cat temp >> snr-2q-"$subsubind"-"$subind";

else

  #international hosts
```

```
  ./recver4l >/dev/null &
          ./sender4 "${hosts["$subind"]}" "${files["$subsubind"]}" >/dev/null 2>temp

  #store loss
  cat temp >> lossnum-4l-"$subsubind"-"$subind"

  #store decoded stream
  cp decstream decstream-4l-"$subsubind"-"$subind"-"$ind"

  #compute and store new snr
  ./snr "${files["$subsubind"]} decstream > temp
  cat temp >> snr-4l-"$subsubind"-"$subind"

    #quadratic interpolation
  ./recver4q >/dev/null &
          ./sender4 "${hosts["$subind"]}" "${files["$subsubind"]}" >/dev/null 2>temp

  cat temp >> lossnum-4q-"$subsubind"-"$subind"

  cp decstream decstream-4q-"$subsubind"-"$subind"-"$ind"

  ./snr "${files["$subsubind"]} decstream > temp
  cat temp >> snr-4q-"$subsubind"-"$subind";
fi
  subsubind=`expr "$subsubind" + 1`
      done
      subind=`expr "$subind" + 1 `
   done

   sleep 30s
   ind=` expr "$ind" + 1 `
done
```