

**KLZ**  
**A Prototype X Protocol Compression System**

Senior Thesis by Ka-Tak Lo  
Supervised by Prof. John M. Danskin

Computer Science Department at Dartmouth College  
Technical Report PCS-TR97-314  
Spring 1997

## **Abstract**

One of the most commonly used graphics protocol is the X Protocol, enabling programs to display graphics images. When running the X Protocol over the network, a lot of structured data (messages with fields) need to be transmitted. Delays can be detected by human users when connected through a low-bandwidth network. The solution is to compress the X protocol. XRemote, a network version of the X Protocol, uses Dictionary-based compression. In XRemote, strings are recorded in the dictionary. When a string repeats, its index in the dictionary is transmitted. Higher Bandwidth X (HBX) uses statistical modeling techniques instead. A context model, which depends on the nature of the field in a particular type of message and the frequencies of the values of the field, is associated with each field. XRemote is much faster than HBX, but HBX achieves better compression than XRemote. The KLZ system is developed to take advantage of our knowledge about the fields in the XMotionNotify packet (what X sends when the mouse moves) and fast Dictionary (LZW) compression. In essence, KLZ reorders and rewrites fields in the XMotionNotify packet so that the fields will be more easily compressed by the fast LZ coder. My experiments show that KLZ compresses this packet nearly as well as HBX, and 5 times better than pure LZ. KLZ is slightly faster than pure LZ, and 10 times faster than HBX. Since many modems already implement LZ compression, KLZ could also be used to reorder data before passing them to the modem with LZ compression for transmission. This reordering would lead to vastly improved compression almost for free.

## **Introduction**

This thesis is about compressing graphics protocols. One may ask why compress graphics protocol in particular? Who will the compression benefit?

### **What is a graphics protocol?**

A graphics protocol is a language that allows an application running on one computer to communicate with the graphics input/output device of another. Suppose a student is working on his/her computer at home and wants to access a program installed on a computer in the school's computer lab. If both computers are running a common graphics protocol, then the student can direct the display of the program running in the school's computer to his/her own computer at home. Whatever the student types will be transferred to the school's computer for processing running with the graphics protocol. Of course both the system on the student's machine and the machine containing the application program have to use the same graphics protocol language in order for them to communicate.

### **Use of the network:**

Why are graphics protocols important? With the development of computer networks, people can run programs on remote machines. There are many instances where one will need to run programs on one computer and display their output on another, or obtain input from another. The major issue here is resource sharing. People should not be prevented from accessing programs that are 100 miles away. One goal that evolved from the idea of resource sharing is to provide high reliability by having alternative sources of supply. For example, files could be duplicated on multiple

machines. If one of the computers goes out of order, there will still be a backup copy on other computers. For military, banking, air traffic control, scientific research and development, the ability to continue operating in face of hardware failure is very important. Another goal is to save money. Small computers have a much better price/performance ratio than large ones. Supercomputers are roughly ten times faster than personal computers but they cost a thousand times more.<sup>1</sup> It is more economical for a computer science student to buy a personal computer with a network connection or a financial analyst to buy a portable laptop so that he or she can do analysis on their company's mainframe and have the result displayed on their personal computers. However, in order for the display to be sent from one computer to another, we need a graphics protocol language embedded in those programs. Therefore, the graphics protocol is an essential part of network computing.

### **Data Compression:**

When do we need data compression in a network system? Data compression is useful in certain situations. If the user is connected with the Ethernet then compression is not very useful since data transmission is very fast. Compression would slow down the transmission by adding computation time. If the application takes more time to generate graphics commands than the network takes to transmit data then compression is not necessary.

However, some networks such as wireless infrared connection, telephone line, or cellular phone, are extremely slow compared to the average speed of the computer used today. When applications are interactive such as video conferencing, web browsing, etc. delays are discernible and cannot be

tolerated by human users. Although fiber optics systems are the fastest medium so far in the computing industry, most telephone networks are composed of twisted pairs made of copper wire. To change the material in all of the world's telephone networks to fiber optics would be very expensive since the cost of labor will be enormous. Therefore, it is reasonable to believe that the growth of processor speed will be greater than the growth of transmission speed of the existing telephone networks.

### **Compressing Graphics Protocol**

The user running a program on a fast remote computer that requires graphics display will not tolerate noticeable delay on a slow network. In order to better utilize the computer while it is transmitting data through the network, we can use the idle computer time to make the size of the data, which are essentially graphics protocol packets, smaller in order to send more output at one time. This way we can better utilize the available bandwidth as well as the idle time of a computer. Furthermore, graphics protocol messages are structured data with fields that are generally predictable. For instance, fields such as window id, user id, etc. will be the same for every graphics packet for a running program. Due to the predictability of the data we can be sure that there's nothing to lose if we omit them from transmission once the first packet with the essential information is sent. As long as both sides of the network use the same compression technique we can guarantee that our data will be correctly deciphered .

### **Text-based Compression Techniques**

Text-based compression depends on the frequency of characters. A frequent character is given a shorter code while an infrequent character has a

longer code. The Huffman Code algorithm is a way to generate a set of codes given the characters and their probabilities. The ideal length of the code for a character should be  $-\log_2 p$ , where  $p$  is the probability of the character. If the character  $t$  occurs 4 times in a string of length 16 ( $p = 1/4$ ) then the code for  $t$  should be 2 bits ( $-\log_2 .25$ ). The length of the code will be exact only when the probability is an inverse power of two, otherwise it will be approximate.

Arithmetic encoding allows the expression of characters in fractional bits.

The most commonly used text compression techniques are dictionary coding and statistical coding.

### **Dictionary Coding**

Words in English text often repeat many times. For instance, in this thesis, the word "the" appears most frequently. One of the techniques that takes advantage of the redundancy of strings is dictionary coding. The most well known example of dictionary coding is LZW. LZ coding was developed as LZ78 by Lempel and Ziv in 1978 and popularized and named by Welch in 1984. LZ78 maintains a dictionary of strings, a prefix string and a new character. When a new character comes in, if the concatenation of it and the prefix string is in the dictionary then the prefix string becomes the concatenation, and there is no output. If the concatenation is not in the dictionary then the index of the prefix string in the dictionary is transmitted, and the prefix string becomes the new character.

### **Statistical Modeling**

In statistical coding, probabilities are assigned to each character based on its frequency in some context. This probability is then used to generate a code. In a zero order coder, like dictionary coding, there is no context. In a first

order coder, the predictions are made based on the previous character which is remembered by the coder. The key to statistical modeling is context. With a given context, codes can be generated based on conditions that are already known. For example, if we are to parse this thesis, the zero order coder will assign a long code to 'Z' because it occurs less frequently than the other 25 English characters. On the other hand, a first order coder will assign a short code to 'Z' because after seeing 'L', we can predict the next letter to be 'Z'. Therefore in the context model of 'L' there is a high probability for 'Z'.

In a straightforward implementation, a zero order model is composed of a table of 256 frequencies and a total character count. Frequencies and total count are important because the probability of a character in a zero order model is the frequency of the character divided by the total count. An implementation of the first order model involves 256 zero order models one for each possible previous character. After the first order model, straightforward implementation is not suitable because it involves  $28^{*(n+1)}$  frequency entries for an  $n^{\text{th}}$  order model. This will use up too much memory. One way to resolve this problem in higher order models is to assign low probabilities to infrequent events, since infrequent events are not very important.

### **Comparison of the Statistical Coding and LZ Coding**

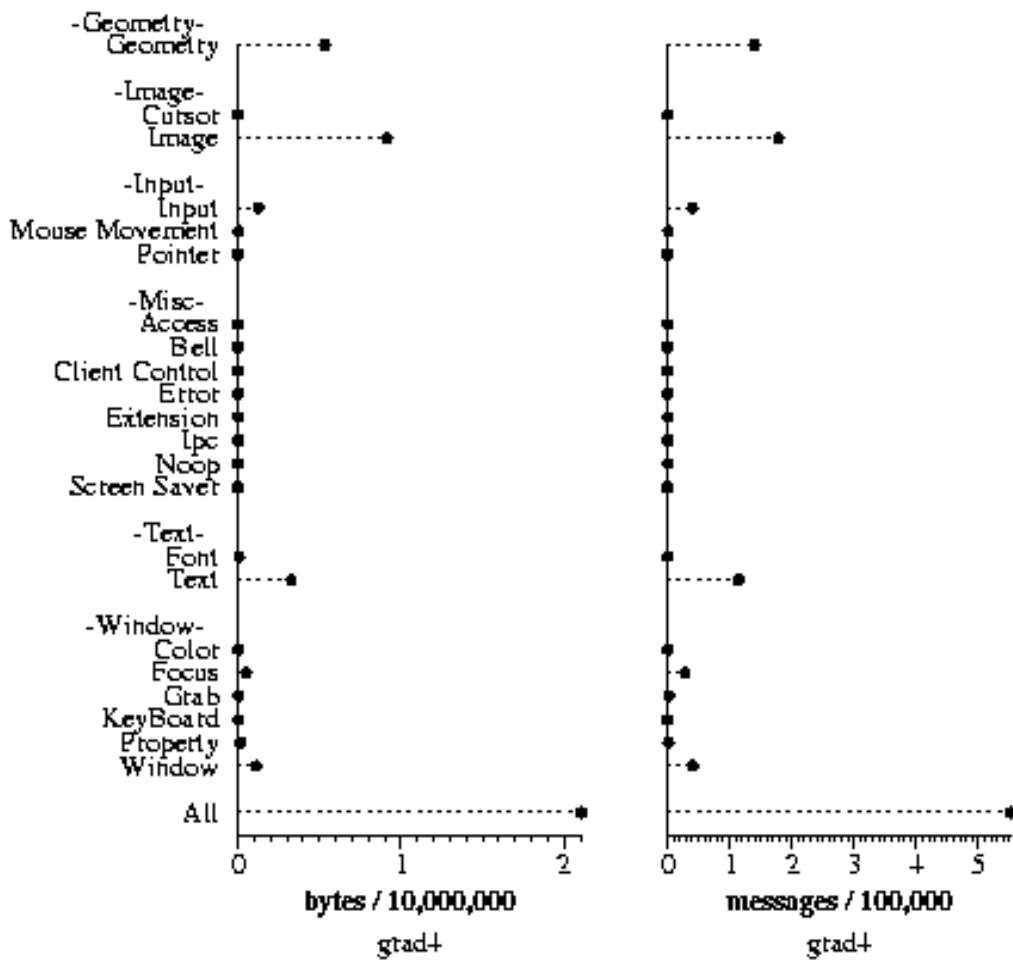
While the LZ compression technique is hardwired to the dictionary algorithm, its performance relies heavily on the occurrences of repeated substrings. However, sometimes data can be completely predictable and lack any common substring within a stream of bytes. The Statistical Modeling method can generate shorter codes and are more flexible in terms of choosing the context. However, it is slow in finding the corresponding context because

it has to look through layers of context models for each input character. The LZ coder is extremely fast due to the simple algorithm of matching strings with entries in the dictionary. The compression ratio in the LZ coder however is not as good as the Statistical coder. Typical performance of the LZ78 coder for English text is 3.5 bits of output per character<sup>2</sup>, whereas PPMC, a third order statistical model, puts out 2.5 bits per character<sup>3</sup>.

### **Goal of this thesis**

The goal of this thesis is to compress the X protocol's mouse motion packets using the LZ method. The reason why this will be useful in experimentation is that mouse motion is an essential part of a graphics protocol. For example, in a graphics editor program, a large number of mouse motion packets are transmitted to tell the client program about the current mouse position. This applies to web browser programs, too. Figure 1 below shows that the -Input category is one of the most frequent message type in a trace of the activities of a computer science graduate student. The X MotionNotify message under the Input message category accounts for a great number of their appearance in the trace. See Appendix A for the a listing of the different types of X messages and the category that they belong to.

Before compressing the packet, it will be restructured in such a way so that the LZ coder will recognize the similarities between the data in the current packet and those are compressed. The messages from the X protocol are structured data, meaning that they consist of fields and each field has its own context. However, the LZ coder will not be able to use this context like the Statistical Coder because it relies on repeated strings. Therefore, we need to find a way to reformat the fields so that a relation between adjacent fields will by recognized by the LZ coder. By taking advantage of the speed of the LZ



Detailed Application Signatures (bytes and messages), trace=grad4

Figure: Messages are categorized according to Appendix A.

coder and the context that is already known about the packets, I hope to achieve fast and excellent compression. Furthermore, since some modems use the V.42 bis scheme<sup>4</sup> which is based on the LZ algorithm to compress data, the idea of this thesis may be transformed into practical use. For example, in order to achieve better compression in a slow network medium, a computer, usually a PC, with the UNIX system which supports the X protocol and using a modem for network connection, can run a program that will reformat the packets before they are sent to the modem for transmission.

## **Previous Work**

### **XRemote**

XRemote is the network version of the X protocol. It uses the delta compression and the LZ compression method to compress X packets. The delta encoder remembers the last 16 messages of length less than or equal to 64 bytes which is typical of most X messages. Incoming messages are compared to the cached messages of the same length. If there are fewer than 8 bytes difference between the input packet and the cached packet of the same length then the delta message is sent instead of the original. A delta message is composed of a one byte opcode, 4 bits selecting one of the 16 cached messages, a 4 bit counter and a list of two byte update fields. The first byte of the update field is the offset of the differing byte in the packet and the second byte is the new value for that byte.<sup>5</sup>

The LZ coder in XRemote outputs the indices of the symbol in the dictionary. The initial dictionary contains 256 possible byte values and a CLEAR symbol at index 257. Whenever an entry needs to be made which requires a 13 bit index in the dictionary, a CLEAR symbol is sent, and the encoder and decoder reset their dictionaries. In addition, whenever the LZ coder outputs a chunk of more than 120 bytes the uncompressed chunk is sent instead, and both the encoder and decoder reset their dictionaries.<sup>6</sup>

### **Result of XRemote simulation**

XRemote compression on geometry and images was extremely bad. On the Idraw and Xfig traces compression is well under 2:1. The X11perf benchmark is extremely repetitive but XRemote only achieves 2:1 compression. A fax protocol regularly achieves 10:1 compression but a ghostscript trace gives less than 3:1 compression. Geometry and images are

important parts of the graphics protocol and yet XRemote performs poorly on them.<sup>7</sup>

### **Structured Data Compression**

A dictionary compressor will not work well with structured data such as graphics protocol messages, object codes, and databases. Structured data can not be well predicted based on the last few bytes which are what the dictionary compressor requires. Each field of the structured data packet needs to be predicted based on the same field in the previous packet. The type of the packet and field involved determine the context of the data to be compressed. Structured data compression uses both Arithmetic coding and Statistical modeling to achieve high compression ratios. The sender and the receiver both have a predictive model that generates a table of probabilities for the next input symbol. The Arithmetic coder on the sender's side uses the table to find the probability of the input symbol  $s_i$ . Then the sender communicates the symbol in  $-\log_2(p_i)$  bits, where  $p_i$  is the estimated probability of the symbol  $s_i$  appearing at this position in the input, to the receiver. The Arithmetic decoder in the receiver's side decodes the bits using its identical table of probability as an inverse map. Individual messages can be expressed in fractional bits. For example, if  $p_i = .75$  then only about .41 bits are needed to update the receiver. With Arithmetic coding the predictive model can be built in a way that is suitable for the data. Unlike LZ compression which is strictly based on repeated substrings, Arithmetic coding is more flexible since the probabilities assigned to symbols are determined by the implementor instead of by the algorithm.<sup>8</sup>

## **Preconditioning of Data**

Structured data need to be preconditioned before they are compressed in order to increase the likelihood of reappearing values in the fields. Absolute values in some fields such as timestamps, and exact mouse coordinates are unlikely to reappear in subsequent packets. Therefore, in order to predict these fields, relative values can be used instead of absolute values. The new data will hold the offset of a field in the current packet from same field in the previous packet. For example, if the time difference is 1 most of the time, then we can give a higher probability to 1. Taking the difference of values sometimes reduces the magnitude of the numbers involved. Predicting mouse coordinates using linear extrapolation on the last two mouse locations, and encoding the difference helps even more. For geometric shapes, each coordinate can be predicted with a third order context based on the previous three coordinates so that if the object is redrawn it will be well predicted.<sup>9</sup>

## **Implementation of Structured Data Compression - HBX**

Higher Bandwidth X (HBX) is a compressed X protocol intended to provide improved interactive performance across low bandwidth interconnections such as serial lines.<sup>10</sup> HBX uses the structured data compression technique discussed above to transmit data from the X client to X server via the pseudo client and pseudo server that the HBX set up in between. The X client communicates with the pseudo server (HBX client), and the X server communicates with the pseudo client (HBX server). The HBX server and client are the compression engines which compress and decompress the data gotten from the X client and the X server. Traffic from multiple streams are parsed into individual messages tagged with message

type, and multiplexed together. The encoding module compresses messages and precedes them with a length field. The decoding module will not start decompressing until the whole message is received.

### **Predictive Models in HBX**

In HBX, since the value of each field in each X message type may have different distributions, a context specifier is used to index a predictive model with each field in each X message type. A good predictor for each field is used to optimize compression of structured data. Since the X protocol supports text, geometry and images, each predictive model is built with a different method. For text compression, HBX implements the PPMC method<sup>11</sup> which uses a hierarchical predictive model that can blend probabilities from models of orders three through zero. For geometry compression, HBX transforms the coordinates into relative coordinates. Then it predicts the relative values using a third order model similar to text compression (i.e. each X coordinate is predicted by the last three X coordinates). Image compression is divided into two levels: small images and large images. A third order hierarchical model is used to predict small images. Large images are compressed pixel by pixel using a set of previous pixels to provide context for the predictive models. Predictive models cannot effectively predict incrementing fields such as timestamps and sequence numbers. As a result, relative values for these fields are used so that they will appear more frequently.<sup>12</sup>

### **Performance of HBX**

HBX's average performance is over 3 times that of XRemote. The relative difference is apparent for geometry and images as in Idraw, Xfig, X11perf and Ghostscript traces. Most significantly, on the X11perf trace, HBX

is able to recognize the repetitiveness of the data and achieves a 216:1 compression ratio. In a ten second excerpt from a Mult trace, HBX achieves a 10:1 ratio for the MotionNotify message. When the bandwidth reaches a peak of 80K bits/sec of uncompressed X traffic, HBX attains a compressed X traffic of 10K bits/sec.<sup>13</sup>

### **Comparison of HBX and XRemote**

Even though HBX has a much higher compression ratio than XRemote, it takes much longer to compress. The Arithmetic coder involves a lot of computation to produce a code based on the probability generated by the predictive models. The two processes together contributes to the relative slowness of HBX. In order to decrease the amount of time used in computation and achieve a higher compression ratio than XRemote, this thesis is going to take advantage of the consistency of some fields in one type of X message, and use the LZ compression technique to compress the preconditioned packet. The combination of the two will result in better compression than XRemote. Furthermore, using the LZ method will potentially make the experiments in this thesis faster than Arithmetic coding.

# Method

## Compression of X protocol using LZ

This thesis will concentrate on using the LZ compression method to compress X protocol packets. In particular, it will compress the MotionNotify packet which is used to communicate the mouse motion. As stated in the introduction, the MotionNotify message is one of the most important types of message that the server sends to the client in a sample trace of the X session. In addition to the data in Figure 1, I had also done some analysis on traces of xfig and netscape sessions. In Figure 2 and 3 below we see that MotionNotify constitutes about 50% of each files.

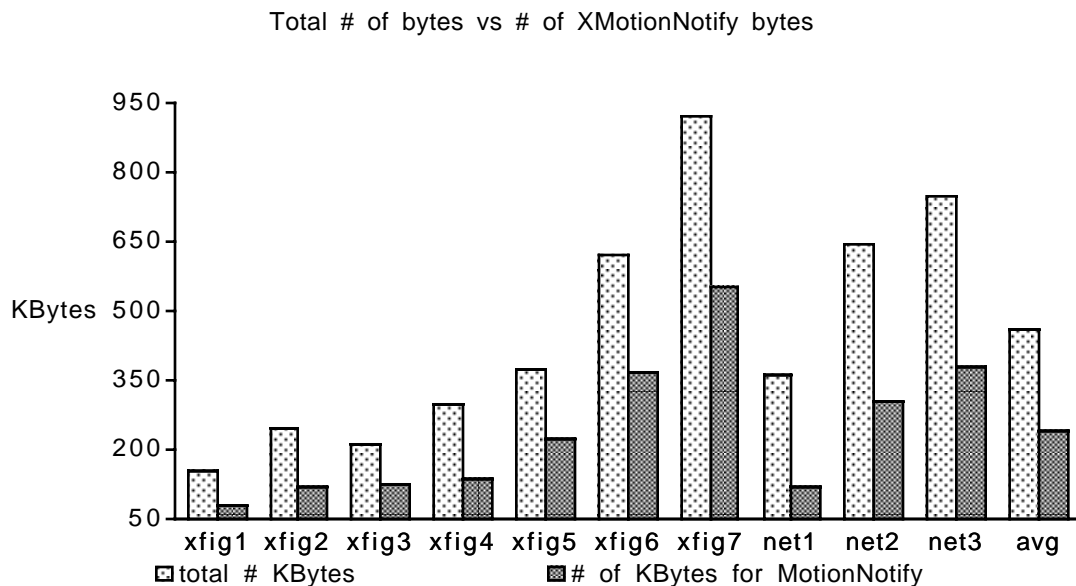


Figure 2: The total number of bytes in each trace file and the number of bytes XMotionNotify takes up in each file. In each case, XMotionNotify takes up about 1/2 total number of bytes which means that XMotionNotify constitutes a significant portion of the traces.

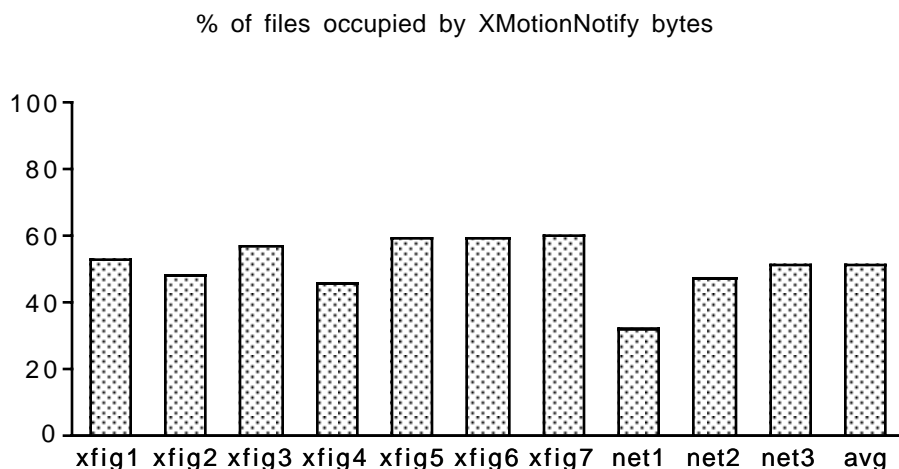


Figure 3: Percentage of files taken by XMotionNotify packets. About 50% of most files are composed of bytes from MotionNotify packets. This shows that they constitute a significant portion of the trace files.

Another reason for choosing this type of X packets is that since we understand the meaning of each field, we can manipulate them in such a way so that there is a repetition of byte pattern that can be found in the dictionary. Table 1 shows the content of the MotionNotify message.

As we can see most of the fields in the packet do not change. Fields such as opcode, detail, same screen, key state are constant for every packet. For event window, root window and child window, they will be constant as long as the mouse is within the same active window throughout the session. For instance, when a user is working on a drawing in one window, the window related fields will be constant. They will only change when the user switch to another window.

The variability of fields in a MotionNotify packet plays a role in constructing a stream of byte pattern that frequently occurs so that a short code can be assigned to the byte stream. This idea of repeating streams is the backbone of the LZ protocol. This thesis will use the LZ protocol to

Field	bytes	Variability
opcode	1	constant
detail	1	constant
seq no	2	variable
timestamp	4	variable
root window	4	most likely constant
event window	4	most likely constant
child window	4	most likely constant
root x	2	variable
root y	2	variable
event x	2	variable
event y	2	variable
key state	2	constant
same screen	1	constant
unused field	1	random

Table 1: Format of the X MotionNotify packet. The field column contains the fields in their respective positions in the packet. The byte column contains the number of bytes in each field. The last column shows the likelihood for each field to be constant or variable.

compress the reformatted X MotionNotify packet to achieve better compression.

### Description of the Experiments

The goal of each experiment is to create a byte stream as long as possible that is most likely to repeat itself. Since the data we are compressing are structured data, presumably the absolute value of a field will not relate to the other fields in the same packet. However, by experimenting the different ways of rearranging the fields we can get an idea of what each field means within the context of a single packet. A few techniques is being used to illustrate this relationship (relationship here means whether or not the fields can be predicted based on previous fields in the same packet.) First, relative values will be used instead of absolute values. As we know LZ can only recognize repeated strings. Therefore, absolute values of some of the fields

like timestamp will appear as a random number to LZ because it can not recognize it from past records. A second technique is to rearrange the fields in the packet so that they will relate to the preceding fields. The combination of the two techniques in a way resembles the method used by HBX. Here we are using Dictionary-based compression and also taking advantage of our knowledge about the context of the fields.

### Experiment I:

The goal of this experiment is to move all of the constant fields to the front of the packet. First, I have to find out which fields remain the same through one trace of the file containing only X MotionNotify packets. I wrote a program that will print out each packet for examination. Many fields remain constant, and they are: code, detail, root, event, child, state, same screen. The remaining fields are variable. After putting the constant fields together and the variable fields after them, Table 2 shows the new

Field	bytes
opcode	1
detail	1
root window	4
event window	4
child window	4
key state	2
same screen	1
time	4
seq no	2
root_x	2
root_y	2
event_x	2
event_y	2

Table 2: The reformatted MotionNotify packet for Experiments I and II. Sample test runs with the first 5 packets of the Xfig trace can be found in Appendix B.

format of the packet. (Note: the unused field is not transmitted because it is not used. The subsequent experiments will not include this field.)

### **Experiment II:**

The last experiment should improve the compression ratio because it creates a string of constant fields in each packet. However, it did not take advantage of the relationship between the previous packet and the current packet. The second experiment will replace the original value of a field with its offset from the same field in the previous packet. Since X MotionNotify tracks mouse motions, the x and y coordinates in the current packet should be very close to those in the previous packet. The differences should be a small number and the pair of offsets is more likely to repeat in the future than the actual coordinates. Therefore, if we attach this pair to the existing byte stream formed by the constant fields, then we might be able to generate a longer byte stream that is likely to appear again in the future. The same holds true for the time field. The packets are recorded in a regular interval so that it is very likely for the offset to repeat. These variable fields are arranged in the same order as the first experiment. See Table 2 for the format of the packet.

### **Experiment III:**

This experiment takes advantage of both the first and the second experiment in that the constant fields are situated at the beginning of the packet and the offset of the variable field is used instead of the actual value. However, in order to increase the occurrence of repeated byte streams, this experiment tries to find a relationship between the offset fields in the packet. From the second experiment, we found the offsets for event\_x and root\_x to be the same. So are event\_y and root\_y. Even though their actual values

Field	bytes
opcode	1
detail	1
root window	4
event window	4
child window	4
key state	2
same screen	1
time	4
root_x	2
event_x	2
root_y	2
event_y	2
seq no	2

Table 3: New format of MotionNotify packet in Experiment III. root\_x and event\_x are put next to each other, similarly root\_y and event\_y are together, too. Seq no field is moved to the end because there is no visible pattern of repetition.

are not the same their offsets are. Therefore we can put them next to each other to make a longer pattern. This will make a shorter code because the two fields are now regarded as a single value in the LZ dictionary entry rather than separate values. The seq\_number field is a little tricky because there does not seem to be any logical pattern for its offset. Therefore, it is moved to the last field. The new format of the packet is shown in Table 3 and the results from a test run are shown in Appendix B.

#### **Experiment IV:**

This final experiment takes all of the advantages found from the previous three experiments. The goal is to create an even longer constant string than the previous experiments. A consistency is found between the offsets of event x and root x, similarly for the y's too, in the two previous experiments. Since the event\_x and the root\_x offset are the same, this implies that the difference between event\_x and root\_x is constant.

Therefore, I assume that we can generate two constant fields that will provide information about the actual event x and event y fields without transmitting their relative offset or their actual values. If we take  $\text{actual\_root\_x} + (\text{last\_root\_x} - \text{event\_x}) - \text{actual\_event\_x} = \text{new\_event\_x}$ , the resulting  $\text{new\_event\_x}$  should be a constant number except for the first time when we have not yet recorded the last difference. This holds for event y and root y as well. If we take the  $\text{new\_event\_x}$  and  $\text{new\_event\_y}$  fields, and put them next to the constant fields then we can potentially create a longer constant string. The decoder at the other end of the network will decompress using the inverse of the equation, namely  $\text{actual\_event\_x} = \text{root\_x} + (\text{last\_root\_x} - \text{event\_x}) - \text{received\_event\_x}$ .  $\text{Received\_event\_x}$  is the value that the decoder receives in the  $\text{event\_x}$  position of the packet. Table 4 shows the new format of the packet. Sample test runs can be found in Appendix B.

Field	bytes
opcode	1
detail	1
root window	4
event window	4
child window	4
key state	2
same screen	1
event_x	2
event_y	2
time	4
root_x	2
root_y	2
seq no	2

Table 4: New format for MotionNotify packet in Experiment IV.  $\text{event\_x}$  and  $\text{event\_y}$  are moved next to the constant byte stream.

## **Summary of Experiments**

From each experiment we are able to understand more about each field in the context of the same and the previous packet. As we proceed, the constant fields and the offset of each field from the previous packet become apparent. Taking the offsets of the event\_x and root\_x fields into account, we can create a new field that is most likely to be constant. By putting the new\_event\_x field next to the stream of constant fields, we can create a longer repeated stream. Therefore, Experiment IV which combines all the advantages from the previous three experiments should achieve better compression than the others.

## **Sources of Test Data**

The test files for this thesis were generated using the hbxS and hbxC<sup>14</sup> programs which generate traces of an X client-server session. Through these programs I was able to generate a file filled with X packets that were being transmitted from the X Client to the X Server, and vice versa. The hbx programs function as a filter between the X Client and the X Server. hbxS runs on the X Server's side of the communication, and the hbxC runs on the machine where the X Client is invoked. When the acX flag is turned on, hbxS will compress all of the packets going from the X Server to hbxC. hbxC will decompress the packets received and then pass the decompressed packet along to the X Client. The reversed process applies to the communication originated from X Client to X Server. When the acX flag is turned off and the dump flag is turned on, two trace files will be generated, each containing all the packets being sent from the server to client and vice versa. The transcript header encapsulating each packet provides information about the type of message the packet contains. With the use of a parser program, I will be able

to extract all the X MotionNotify packets from a trace and write the MotionNotify packets onto a partial trace file.

## Tests

All experiments were ran on DEC Alphas. As mentioned before, a series of trace files were generated using the hbxC and hbxS programs. A parser program extracted all the XMotionNotify packets from each trace and deposit them into the partial trace file. There are 7 partial traces for each xfig sessions, and 3 netscape traces. They are named as xfig1, xfig2, ... , xfig7, and net1, net2, and net3. The five experiments are pure LZ - the unmodified LZ protocol, and the four modified versions of LZ: KLZv1, KLZv2, KLZv3, and KLZv4. In addition, the same trace files are also ran with the tran\_hbx program, which is a variation of the hbx programs which takes transcript or trace files as input. It compresses and decompresses each input file using the Arithmetic coding method in HBX. The acX flag is turned on in order to perform the compression. Each test was ran with the /bin/time function which measures the amount of real time, user time, and system time used by each test run. Running each experiment on each file 5 times allowed us to measure the average time it took to compress and decompress a file. One difficulty worth noting is that the running time for a test on one computer may be different from the same test on another computer. However, since I ran all the tests on the same computer, there was no inconsistency in the results.

# Results

The four experiments of this thesis prove to accomplish much better compression than the original LZ protocol. However, it is not as good as the HBX protocol. The compression ration of the HBX is slightly higher than KLZv4. Although the compression ratio is not as good, on the average, KLZv4 is 14 times more efficient than HBX when measured in real time.

## Compression

Figure 4, shows the output as a percent of input for each of the LZ experiments performed on each file. Figure 5 below is derived from the same data as those for Figure 4 except that the compression ratio is shown instead.

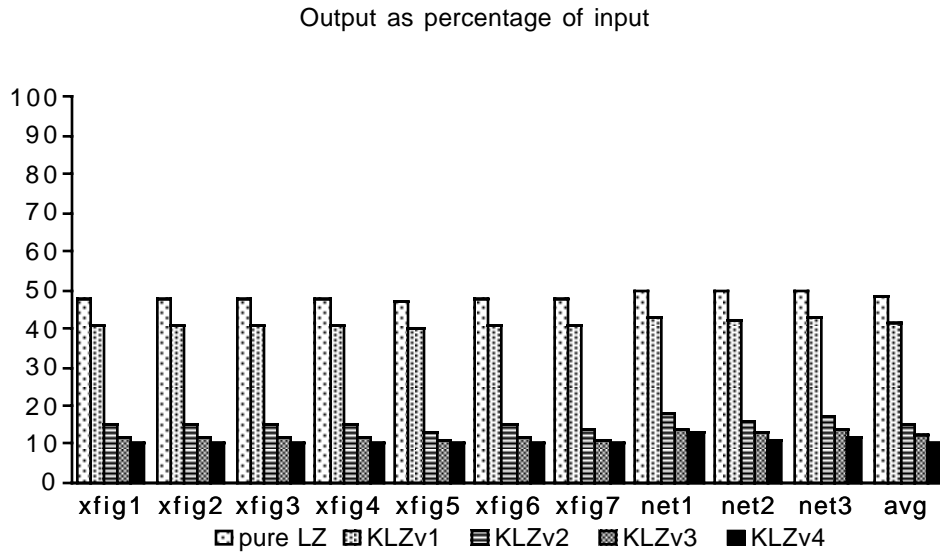


Figure 4: Compression performance of the five versions of the LZ compression method acting on the X MotionNotify packet. The pure LZ protocol achieves a 2:1 compression ratio. KLZv1 presents a slight improvement of about 20% over pure LZ. The later three versions which takes the offset instead of the absolute values of each field improve dramatically. KLZv4 is 5 times better than pure LZ. This is a significant increase in terms of performance.

On the average, pure LZ outputs half of its input which is not very impressive. KLZv1 is about 20% better than pure LZ. As we progress from KLZv2 to KLZv4, the result improves drastically. KLZv2, KLZv3 and KLZv4 use the offsets or relative values in the fields which are likely to be small integers. Therefore, taking relative values proves to be more efficient in Dictionary-based compression than the actual values which rarely repeat. By far KLZv4 outperforms the other versions because it not only takes advantage of using relative values but it also creates a longer constant stream at the beginning of a packet. Therefore, it accomplishes better compression than the rest.

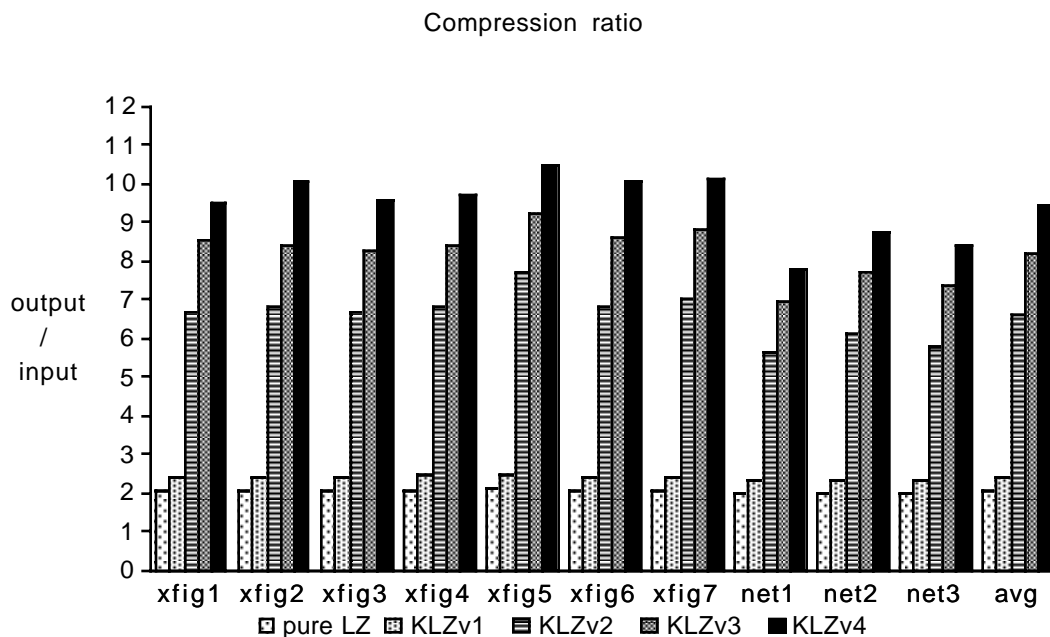


Figure 5: Compression ratio of xfig and netscape traces using the five versions of the LZ protocol. The original protocol outputs one byte for every 2 bytes of input. The KLZv1 is not much better than the original. However, in KLZv2 there is a great increase in ratio, one byte of output for every 7 bytes of input. The ratio is increased by one in each of the last two versions which are variation of KLZv2 using the offset idea combined with the rearrangement of fields. The best version, KLZv4 accomplishes about 9:1 ratio, which is 4.5 times better than the pure LZ protocol.

Figure 6 shows the compression ratio of LZ, AcX and KLZv4. AcX represents Arithmetic coding with Predictive modeling used by HBX (tran\_hbx in this case). AcX achieves about 12:1 compression on the average, which is one third better than the 9:1 compression of KLZv4. This is due to the fact that AcX uses the structured data compression to its full potential when generating code for each field, whereas KLZv4 takes a while for the long constant string to be registered in the dictionary. Therefore long codes are produced by KLZv4 at the beginning of the process. Whenever a 13 bit code is needed the dictionary will be reset. During this rebuilding process, long codes are generated. The highest compression for AcX is 13:1 whereas the highest for LZ is just 2:1. The highest for KLZv4 is 10:1.

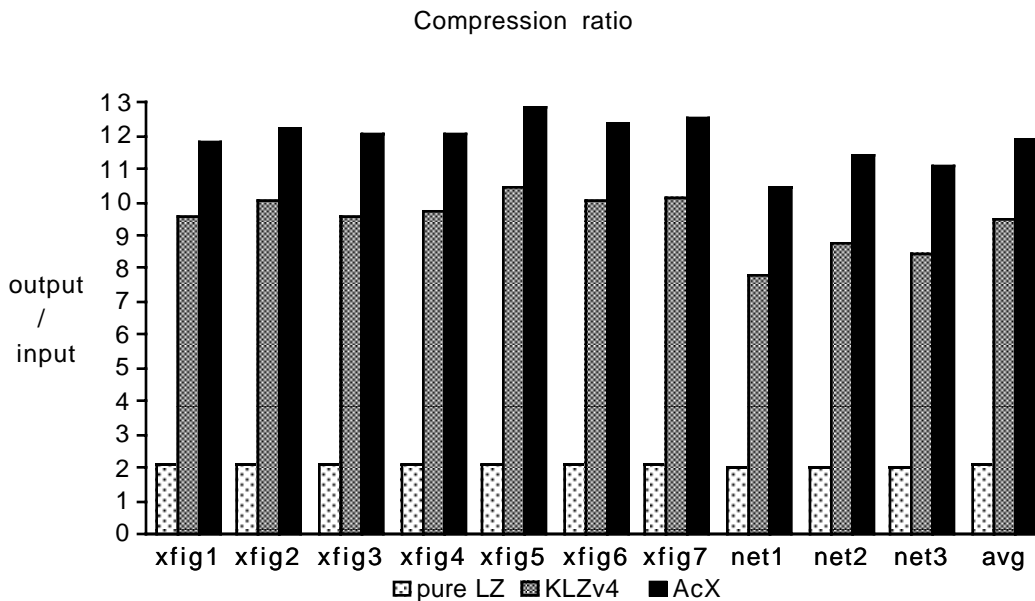


Figure 6: Compression ratios of the three compression techniques: LZ, AcX (Arithmetic Coding in HBX), and KLZv4. The highest compression ratio is achieved by AcX, about 13:1. KLZv4's highest ratio is 10:1. This is expected since AcX is specially designed to compress X messages whereas KLZv4 is trying to take advantage of the nature of the data but is restricted by the dictionary compression technique to recognize repeated strings. LZ by far is the worst of the three, it compresses 5 to 6 times less than the other two protocols.

Output as percentage of Input

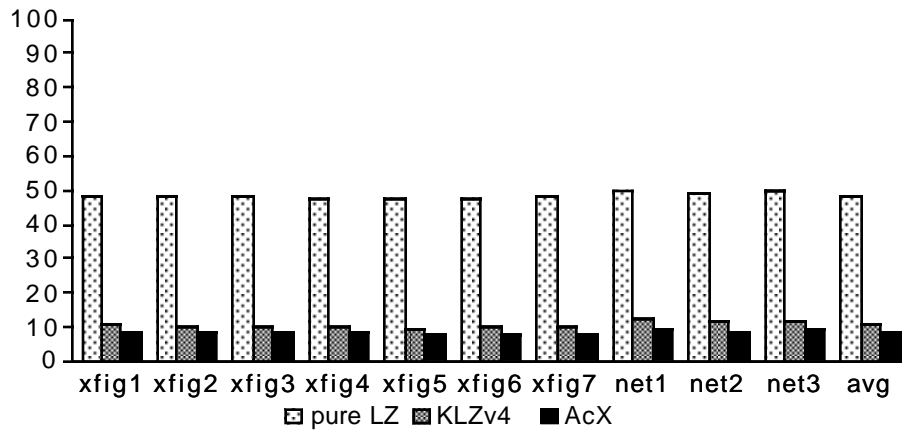


Figure 7: Output as a percentage of input for the three protocols: LZ, AcX, and KLZv4. The percentages in this graph presents a more accurate measure of the compression performance. Even though AcX is 1/3 better than KLZv4 in terms of compression ratio, that amount only accounts for 1-2% difference in the amount of output. The compression ratio of the netscape traces are generally less than the xfig traces in Figure 6. However, in terms of percentage that difference is fairly minute. The vast difference between LZ and the other two protocols can easily be detected in this figure.

Figure 7 shows output as percentage of input for the original LZ, AcX, and KLZv4. From Figure 6 we know that AcX is about 1/3 better than KLZv4. Surprisingly, that amount only accounts for 1-2% difference in the percentage of output for the two protocol. Pure LZ has a much higher percentage than the other two protocols. The large amount of output that LZ produces will slow down the decompression rate as we shall see in the next section.

### Compression Rate

Figure 8 shows the compression rate of the different versions of the LZ protocol based on the recorded real time. One might expect the compression speed of the experiments to be slower than pure LZ. However, data show that each experiment except for KLZv1 is slightly faster than the previous ones.

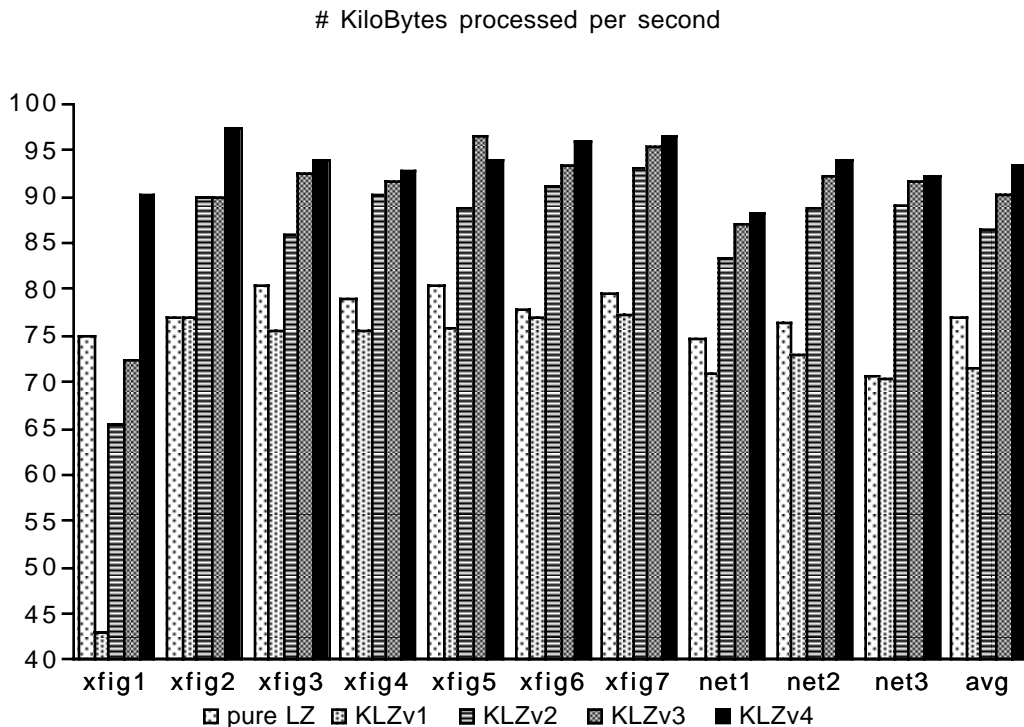


Figure 8: Byte processing rate of the five LZ experiments. Calculation is based on input size and real time recorded with the `/bin/time` function. The time that pure LZ takes to process is surprisingly slightly more than the amount of time KLZv4 takes for every file. The fact that pure LZ is producing much more output than KLZv4 explains the extra time it takes to decompress. This shows that the amount of bytes to decompress dominates the amount of time spent on reformatting each packet, even though KLZv4 does more computation.

As mentioned in Figure 7, pure LZ produces 40% more output than the KLZv4. Therefore, pure LZ takes more time to decompress the packets whereas the LZ experiments output fewer bytes and subsequently spend less time decompressing. This also tells us that the computation time spent on reformatting the packets before compression and putting them back into the original form after decompression, hardly plays a role in the total amount of execution time. If this reformatting were expensive then

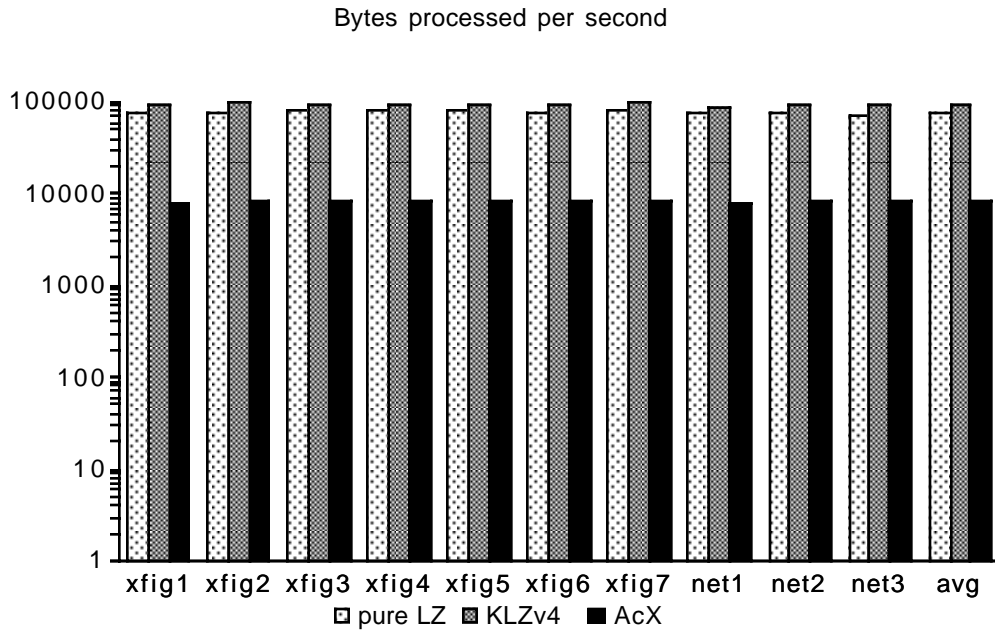


Figure 9: Number of bytes processed per second using the three protocols. The graph is shown in logarithmic scale due to the vast difference between the amount for AcX and the LZ protocols. AcX is about 11 times slower than the other two. In general, pure LZ and KLZv4 can process at most 94 KBps with pure LZ processing fewer bytes than KLZv4, due to the extra output generated by pure LZ.

execution time would increase instead of decrease for the last 3 experiments. Furthermore, the large amount of output that pure LZ generates leads to a considerable amount of searching through the dictionary for every input character. Since KLZv4 achieves a much better compression than pure LZ, about 4.5 times less bytes, the amount of work done by KLZv4 to decompress is relatively less than pure LZ.

Figure 9 and 10 show the process rate of the three protocols: pure LZ, AcX, and KLZv4. Pure LZ processes about 77 KBps whereas KLZv4 processes about 94 KBps, about 1/4 more than pure LZ. Furthermore, Figure 10 shows that pure LZ is outputting about 4 times as many bytes per second as KLZv4. For KLZv4, the network bandwidth only has to be at least 10 KBps for it to

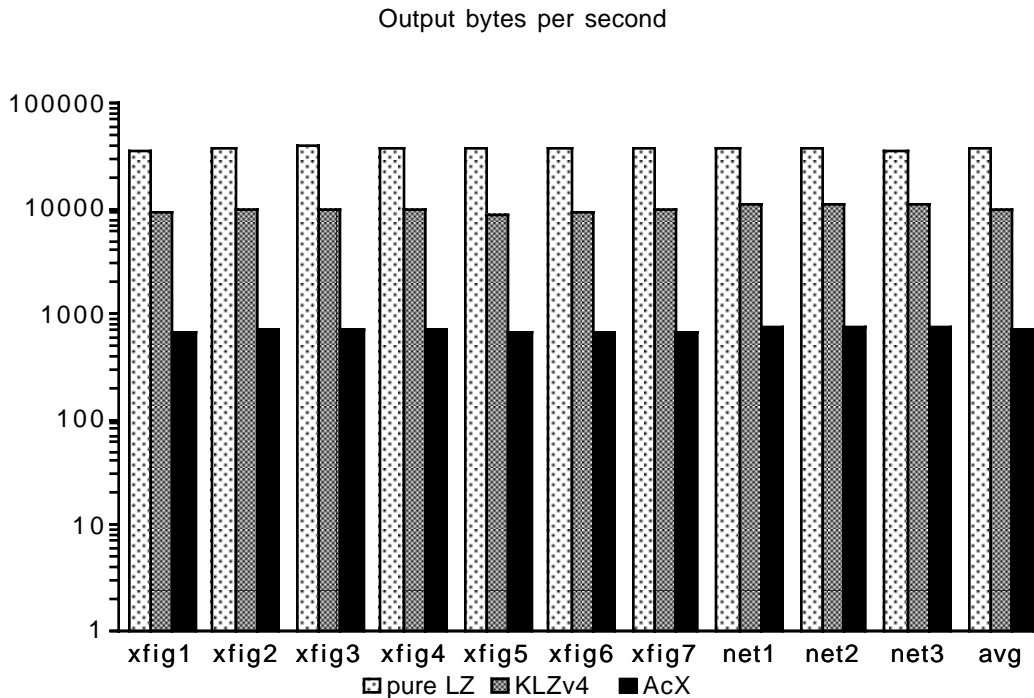


Figure 10: Number of bytes per second output by the three protocols: pure LZ, AcX, and KLZv4, shown in logarithmic scale. This graph in a way resembles the inverse of Figure 9. Pure LZ in Figure 9 process fewer bytes per second than KLZv4. In this graph pure LZ outputs more bytes per second than KLZv4. AcX is again proportional to KLZv4, roughly about 10 times less bytes in output per sec than KLZv4. Therefore, AcX both processes and outputs fewer bytes per second than KLZv4.

benefit from the compression method whereas for pure LZ as much as 37 KBps is needed.

Although AcX achieves a better compression ratio than KLZv4, its compression rate is much poorer than the latter. AcX can process 8.4 KBps. This is about 11 times slower than the process rate of KLZv4(94 KBps). One might ask when will we use AcX? The answer lies in Figure 10 which shows that AcX outputs about 14 times fewer bytes than KLZv4. In order for AcX to be effective we only need a network bandwidth of 700 Bps whereas for KLZv4 we need as much as 10 KBps. However, since AcX takes so long to process the

bytes, it might just be more useful in some situations to use KLZv4 when the network bandwidth is greater than 10 KBps and less than 94 KBps.

Figure 11a-c below: Execution time and range of measured times for pure LZ, AcX, and KLZv4. Execution time is measured in real time using the /bin/time function on the Unix system. The marks on the graph indicate maximum, average and minimum of the time recorded. The upper end of the mark is the maximum; the middle dash represents the average; the lower end is the minimum. The distance between the max and min is very small for every file ran by every protocol. This means that the error is fairly negligible.

Note: 11a, b and 11c have very different scales.

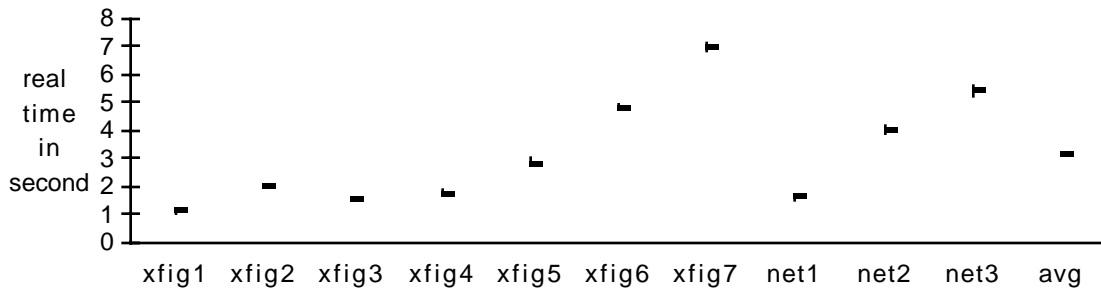


Figure 11a: Execution time of pure LZ and the range of times measured for each file.

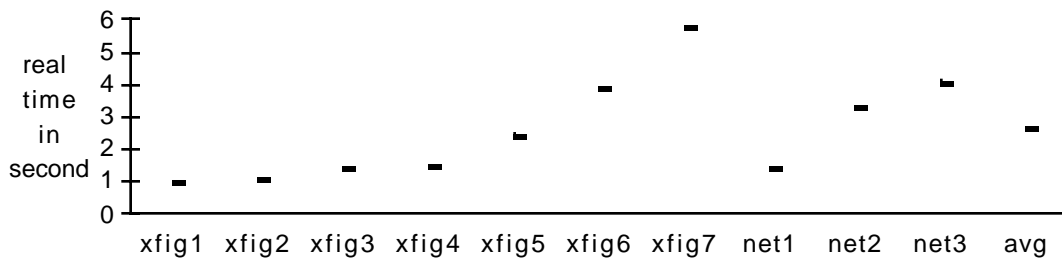


Figure 11b: Execution time of KLZv4 and the range of times measured for each file.

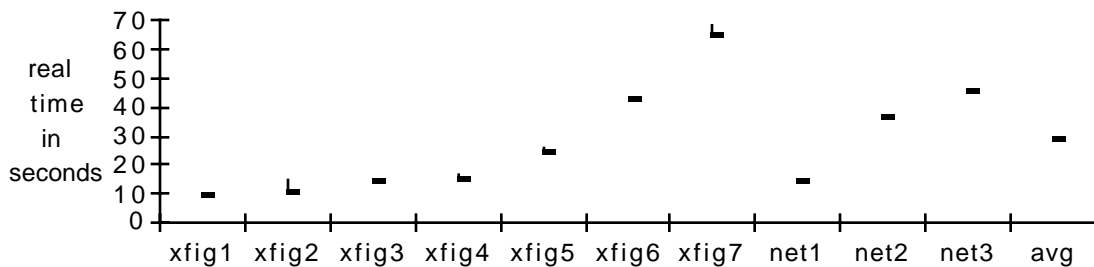


Figure 11c: Execution time of AcX and the range of times measured for each file.

Figure 11 shows the execution time and the corresponding range of measurements for pure LZ, KLZv4 and AcX. The range of execution time of each file over the five runs is relatively small. In the pure LZ experiments the standard deviation is about .10 and KLZv4 is about .05, not bad in terms of degree of deviation. AcX has a little higher standard deviation of .7. This might have to do with manual error during the data entry process. Since the execution time for AcX is large compared to .7, the deviation is negligible as long as we can get an idea of how fast AcX performs, and can draw a valid comparison between KLZv4 and AcX. In general, the tests conducted with the /bin/time function is fairly consistent.

# Discussion

## **Summary of Results**

In terms of compression ratio, KLZv4 delivers three times the performance of pure LZ compression. Not only did KLZv4 produce less output, but KLZv4 is also 20% faster than the original LZ protocol. Apparently, KLZv4 is faster because of tremendous reduction in the size of the compressed data.

In contrast to the poor performance of the original LZ protocol, the AcX protocol is very effective at compressing structured data. AcX achieves on the average a ratio of 12:1, whereas KLZv4 achieves only 9:1. Due to the fact that AcX is especially designed to handle structured data such as messages in the X protocol (AcX is used in the HBX program whose target is to compress X protocol), it can generate shorter code for a field based on the context of that field known from previous packets. In contrast, KLZv4 depends heavily on dictionary coding which sometimes generates longer codes while it is in the process of building the dictionary. Even though we are able to manipulate the fields and make a long constant string in each packet, at the start of the compression and whenever the dictionary needs resetting, it will take a while for the coder to register the constant string in the dictionary. In brief, AcX coding is immediately optimal while LZ coding is asymptotically optimal. This explains KLZv4's relatively poor compression ratio compared to AcX. On the other hand KLZv4 is about 11 times faster than AcX. In order to AcX to be useful, we need a network bandwidth between 700 Bps and 8.4 KBps. The lower bound for AcX is much less than that of KLZv4 which is 10 KBps. However, KLZv4 increases the upper bound to 94 KBps. Therefore, considering the present speed of the network, it might be worth exploring a

faster compression method than a slower one which accomplishes slightly better compression performance.

### **Disclaimer**

The real time gotten from the test runs seems to vary from machine to machine. It depended on when I ran the experiments. The test run was performed during a period when there were few users logged onto the server. The real time on each test run is less than previous test runs that I had performed at another time when more users were logged on. Therefore, the numbers in the compression rate graphs should only serve as a guide to how well each protocol executed at the time of the test. Those numbers should not be used as the exact numbers in determining the type of network (fast network like Ethernet, 33.6, 28.8, 14.4 modems, etc.) to use when running these programs.

# Appendix A

## Taxonomy of X Messages

X messages are categorized functionally. We have a two level hierarchy. Top level categories represented by "Class - category". Second level categories begin with a "\*" and are on the same line with a message. Messages beginning with "X\_" are client requests or queries. Other messages are server events (such as "KeyPress"), "StreamInit" or "Error". "StreamInit" is the connection startup sequence. Server replies to queries are categorized with the queries, and are therefore not listed.

Class - GEOMETRY

\*Geometry

Message

X\_ClearArea  
X\_FillPoly  
X\_PolyArc  
X\_PolyFillArc  
X\_PolyFillRectangle  
X\_PolyLine  
X\_PolyPoint  
X\_PolyRectangle  
X\_PolySegment

Class - IMAGE

\*Cursor

X\_CreateCursor  
X\_CreateGlyphCursor  
X\_FreeCursor  
X\_QueryBestSize  
X\_RecolorCursor  
X\_CopyArea  
X\_CopyPlane  
X\_GetImage  
X\_PutImage

\*Image

Class - INPUT

\*Input

X\_ButtonPress  
X\_ButtonRelease  
X\_EnterNotify  
X\_FocusIn  
X\_FocusOut  
X\_KeyPress  
X\_KeyRelease  
X\_LeaveNotify  
X\_MotionNotify  
X\_GetMotionEvents  
X\_QueryPointer

\*Mouse Movement

	X_TranslateCoords
	X_ChangePointerControl
	X_GetPointerControl
	X_GetPointerMapping
	X_SetPointerMapping
	X_WarpPointer
Class - MISC	
*Access	X_ChangeHosts
	X_ListHosts
	X_SetAccessControl
*Bell	X_Bell
*Client Control	X_KillClient
	X_SetCloseDownMode
*Error	Error
	Extension-request
	X_ListExtensions
	X_QueryExtension
*Ipc	ClientMessage
	ColormapNotify
	SelectionClear
	SelectionNotify
	SelectionRequest
	X_ConvertSelection
	X_GetSelectionOwner
	X_SendEvent
	X_SetSelectionOwner
*Noop	X_NoOperation
*Screen Saver	X_ForceScreenSaver
	X_GetScreenSaver
	X_SetScreenSaver
Class - TEXT	
*Font	X_CloseFont
	X_GetFontPath
	X_ListFonts
	X_ListFontsWithInfo
	X_OpenFont
	X_QueryFont
	X_QueryTextExtents
	X_SetFontPath
*Text	X_ImageText16
	X_ImageText8
	X_PolyText16
	X_PolyText8
Class - WINDOW	
*Color	X_AllocColor
	X_AllocColorCells

	X_AllocColorPlanes
	X_AllocNamedColor
	X_CopyColormapAndFree
	X_CreateColormap
	X_FreeColormap
	X_FreeColors
	X_InstallColormap
	X_ListInstalledColorMaps
	X_LookupColor
	X_QueryColors
	X_StoreColors
	X_StoreNamedColor
	X_UninstallColormap
*Focus	X_GetInputFocus
	X_SetInputFocus
	X_AllowEvents
	X_ChangeActivePointerGrab
	X_GrabButton
	X_GrabKey
	X_GrabKeyboard
	X_GrabPointer
	X_GrabServer
	X_UnGrabButton
	X_UnGrabKey
	X_UnGrabKeyboard
	X_UnGrabPointer
	X_UnGrabServer
*KeyBoard	X_ChangeKeyboardControl
	X_ChangeKeyboardMapping
	X_GetKeyboardControl
	X_GetKeyboardMapping
	X_GetModifierMapping
	X_QueryKeymap
	X_SetModifierMapping
*Property	X_PropertyNotify
	X_ChangeProperty
	X_DeleteProperty
	X_GetAtomName
	X_GetProperty
	X_InternAtom
	X_ListProperties
	X_RotateProperties
*Window	CirculateNotify
	CirculateRequest
	ConfigureNotify
	ConfigureRequest

CreateNotify  
DeleteNotify  
Expose  
GraphicsExpose  
GravityNotify  
KeymapNotify  
MapNotify  
MapRequest  
MappingNotify  
NoExpose  
ReparentNotify  
ResizeRequest  
StreamInit  
UnmapNotify  
VisibilityNotify  
X\_ChangeGC  
X\_ChangeSaveSet  
X\_ChangeWindowAttributes  
X\_ChangeWindow  
X\_ConfigureWindow  
X\_CopyGC  
X\_CreateGC  
X\_CreatePixmap  
X\_CreateWindow  
X\_DestroySubwindows  
X\_DestroyWindow  
X\_FreeGC  
X\_FreePixmap  
X\_GetGeometry  
X\_GetWindowAttributes  
X\_MapSubwindows  
X\_MapWindow  
X\_QueryTree  
X\_ReparentWindow  
X\_SetClipRectangles  
X\_SetDashes  
X\_UnmapSubwindows  
X\_UnmapWindow

## **Appendix B**

Test runs of the four experiments in this thesis on the trace file xfig1.

### **Explanation of fields:**

opcode - type of packet. Since xfig1 contains only XMotionNotify packets, the opcode remains the same through each experiment.

detail - in graphics protocol, this field acts as a flag used by the X Server to tell the X Client how often the Server is sampling the mouse motion.

seq no - every request and response sent between the X Client and Server has a sequence number. The seq no field is incremented whenever a new request is sent from the Client to the Server.

time - the time at which the packet is transmitted by the Client or the Server.

root - the root window id or the screen.

event - the event window id or the active window.

child - the child window id inside the active window.

root x - the x coordinate of the mouse on the root window.

root y - the y coordinate of the mouse on the root window.

event x - the x coordinate of the mouse on the event window.

event y - the x coordinate of the mouse on the event window.

key state - state of the keyboard, flag for whether a key is pressed or not.

same screen - flag for being on the same screen or not.

Field	packet 1	packet 2	packet 3	packet 4	packet 5
opcode	6	6	6	6	6
detail	0	0	0	0	0
seq no	562	562	562	562	562
time	416235301	416235318	416235337	416235354	416235373
root	41	41	41	41	41
event	50331909	50331909	50331909	50331909	50331909
child	0	0	0	0	0
root	41	41	41	41	41
root x	449	542	542	576	605
root y	195	248	248	267	276
event x	183	276	276	310	339
event y	27	80	80	99	108
key state	0	0	0	0	0
samescreen	1	1	1	1	1

Table B1: Table shows the content of the first five packets recorded during a test run with the pure LZ method acting on xfig1.

Field	packet 1	packet 2	packet 3	packet 4	packet 5
opcode	6	6	6	6	6
detail	0	0	0	0	0
root	41	41	41	41	41
event	50331909	50331909	50331909	50331909	50331909
child	0	0	0	0	0
key state	0	0	0	0	0
samescreen	1	1	1	1	1
time	416235301	416235318	416235337	416235354	416235373
seq no	562	562	562	562	562
root x	449	542	542	576	605
root y	195	248	248	267	276
event x	183	276	276	310	339
event y	27	80	80	99	108

Table B2: Table shows the contents of the first five packets during a test run of Experiment I acting on xfig1. All of the fields that are constant through out the test run are placed at the front of the packet. The variable fields follow the constant fields in this experiment.

Field	packet 1	packet 2	packet 3	packet 4	packet 5
opcode	6	6	6	6	6
detail	0	0	0	0	0
root	41	41	41	41	41
event	50331909	50331909	50331909	50331909	50331909
child	0	0	0	0	0
key state	0	0	0	0	0
samescreen	1	1	1	1	1
time	416235301	17	19	17	19
seq no	562	0	0	0	0
root x	449	49	44	34	29
root y	195	29	24	19	9
event x	183	49	44	34	29
event y	27	29	24	19	9

Table B3: Table shows the contents of the first five packets during a test run of Experiment II acting on xfig1. All of the constant fields are placed at the front of the packet. The variable fields are replaced with their offset from the previous packet instead of their actual values. Notice that the offset for the time is reduced to a much smaller number. The coordinates are also reduced to a more smaller number which is likely to repeat later. Also notice that the root x and event x fields are the same after the first packet, as well as the y's.

Field	packet 1	packet 2	packet 3	packet 4	packet 5
opcode	6	6	6	6	6
detail	0	0	0	0	0
root	41	41	41	41	41
event	50331909	50331909	50331909	50331909	50331909
child	0	0	0	0	0
key state	0	0	0	0	0
samescreen	1	1	1	1	1
time	416235301	17	19	17	19
root x	449	49	44	34	29
event x	183	49	44	34	29
root y	195	29	24	19	9
event y	27	29	24	19	9
seq no	562	0	0	0	0

Table B4: Table shows the contents of the first five packets during a test run of Experiment III acting on xfig1. All of the constant fields are placed at the front of the packet followed by the variable fields using their offsets from the previous packet. The ordering of the root x and event x fields are different from Experiment II. Since their offsets are the same after the first packet, they are put together so that the coder will recognize the repetition immediately. Same applies to the y's.

Field	packet 1	packet 2	packet 3	packet 4	packet 5
opcode	6	6	6	6	6
detail	0	0	0	0	0
root	41	41	41	41	41
event	50331909	50331909	50331909	50331909	50331909
child	0	0	0	0	0
key state	0	0	0	0	0
samescreen	1	1	1	1	1
event x	266	532	532	532	532
event y	168	336	336	336	336
time	416235301	17	19	17	19
root x	449	49	44	34	29
root y	195	29	24	19	9
seq no	562	0	0	0	0

Table B5: Table shows the contents of the first five packets during a test run of Experiment IV acting on xfig1. All of the constant fields are placed at the front of the packet. The variable fields are rearranged. After noticing the equality of the root x and event x offset, event x is reworked into a constant field. The new event x = actual root x + (root x - event x in last packet) - actual event x. Same formula applies to event y. The decoder will use the inverse of the equation, event x = actual root x + (root x - event x in last packet) - received event x, to find the actual event x. This way notice that the event x and event y fields being moved to follow the original constant fields creates a longer constant string pattern in each packet.

## EndNotes

1. Tanenbaum, Andrew S. Computer Networks, 3<sup>rd</sup> Edition. (New Jersey: Prentice Hall PTR, 1996), 3.
2. Danskin, John M. Compressing the X Graphics Protocol. (New Jersey: Princeton University, 1994), 14.
3. Danskin, Compressing the X Graphics Protocol, 15.
4. Tanenbaum, Computer Networks, 112.
5. Danskin, Compressing the X Graphics Protocol, 47.
6. Danskin, Compressing the X Graphics Protocol, 50.
7. Danskin, Compressing the X Graphics Protocol, 57.
8. Danskin, Compressing the X Graphics Protocol, 61.
9. Danskin, Compressing the X Graphics Protocol, 67.
10. Danskin, Compressing the X Graphics Protocol, 80.
11. Danskin, Compressing the X Graphics Protocol, 83.
12. Danskin, Compressing the X Graphics Protocol, 82-86.
13. Danskin, Compressing the X Graphics Protocol, 87-88.
14. Danskin, John M. "Higher Bandwidth X Programs", <http://www.cs.dartmouth.edu/~jmd/>

## **Bibliography**

Danskin, John M. Compressing the X Graphics Protocol. New Jersey: Princeton University, 1994.

Tanenbaum, Andrew S. Computer Networks, 3<sup>rd</sup> Edition. New Jersey: Prentice Hall PTR, 1996.