NOMACHINE		NX X Protocol Compression	
Prepared by: Gian Filippo Pinzari		*N°:* D-309/3-NXP-DOC	
Approved by: Gian Filippo Pinzari	*Signature:*	*Date:* 26/09/2003	*Amended:* A

NX X Protocol Compression

NOMACHINE	NX X Protocol Compression		
Prepared by: Gian Filippo Pinzari		*N°:* D-309/3-NXP-DOC	
Approved by: Gian Filippo Pinzari	*Signature:*	*Date:* 26/09/2003	*Amended:* A

Index

NX X Protocol Compression

This document deals with the methods used in the NX project to minimize the network bandwidth needed by X-Window protocol, as used by contemporary Unix desktop applications.

It is a common misconception that X itself is good enough because people can run X terminals even over a modem. A normal X session, browsing the Internet or accessing common desktop applications, generates hundreds of megabytes of protocol data. Efficient compression is not only needed to run applications over slow-bandwidth links, but even to run multiple user sessions on common corporate LANs.

NX's goal is to permit users to run, over the Internet, the same colorful and graphic intensive applications they run on their desktop computers. We don't try to pose any requirement on which X applications can or cannot be executed. To effectively use other remote display solutions, users have to disable fancy backgrounds, drop-down menus' animations and similar graphics effects. NX was designed to deal with such “extreme” conditions without users or desktop applications' developers having to modify their habits or their code. This was, since the beginning, our most important requirement.

Everybody who worked on enabling X-Window protocol to operate over low-bandwidth, high-latency links will tell that the problem doesn't only reside in compression. Infact, NX X compression is just a part (an important part, indeed) of nxproxy work. It is worth noting, anyway, that many X developers consider 384 Kbps or higher ADSL connections, with latency in the order of 50ms, to be in the low-bandwidth category. In NX, when we speak about low-bandwidth, we speak about 9.6 Kbps GSM modems, with latency in the order of 500 ms. An important part of NX design and implementation was dedicated to reducing round-trips and implementing strict flow-control of data traveling through the low-bandwidth proxy link. At the same time, NX could not have reached the present level of performance without a good solution to the X compression problem.

NX X protocol compression is derived from DXPC. The DXPC - Differential X Protocol Compressor project, released in 1995 by Brian Pane, not only was an invaluable source of ideas and a very good base on which NX X compression was started, it also offered specific differential encoding of many of the nearly 160 requests, replies and events that constitute the core X protocol.

Message Store Based X Protocol Encoding

NX considers any X message to be composed of a fixed size part, called identity, and a data part, of variable size.

This is the C language representation of the PolySegment X protocol request from Xproto.h:

```
#define X_PolySegment            66

#define sz_xPolySegmentReq       12

typedef struct {
   CARD8 reqType;
   BYTE pad;
   CARD16 length B16;
   Drawable drawable B32;
   GContext gc B32;
} xPolySegmentReq;
```

The identity part usually corresponds to the X request's structure, while the data part is what follows. In this case the identity part has a length of 12 bytes.

NX maintains in the main memory a cache of the last X messages sent through the wire, divided by protocol opcode. This cache is named MessageStore. To allow a fast look-up of messages in the specific MessageStore, NX calculates a MD5 checksum of any new request or reply that has to be encoded. Any message type has its own method to calculate the MD5 of the identity, while MD5 of the data part is simply obtained by adding any data byte to the checksum. NX maintains information about opcode implicitly, by adding messages to the right MessageStore, and saving size in a specific field of the base Message class.

The MessageStore's method calculating MD5 of identity has to be carefully chosen to not include those fields that are likely to change across different instances of the same X request. In the case of PolySegment, the method calculating MD5 of identity is empty, as both Drawable and GContext are likely to be different in any new request.

When a new PolySegment request is received, NX calculates the checksum of the new message and searches it in the MessageStore. If the message is found, NX only sends this status information to the remote peer, together with the position where the message can be retrieved from store and a differential encoding of all those fields that are not part of the identity checksum.

In case of PolySegment, the fields that need to be differentially encoded are Drawable and GContext. All atomic values that NX sends over the network have a specific encoding, usually based on a integer or character cache. Differential encoding is treated in a separate section.

Encoding of status and position in MessageStore cache requires between 2 and 6 bits, depending on the protocol message. Drawable and GContext (originally 64 bits) can be encoded in as few as 2 to 8 bits. This makes it possible to encode a PolySegment request of 176 bits (22 bytes) in 4 to 14 bits, with a compression ratio of more than 10:1.

If the message cannot be found in cache, proxy encodes the message field by field. It also prepends the position where the decoding side has to store the message in cache. As it is the encoding side to mandate the position in cache, we can say that each proxy manages the cache of its remote peer. In this way, any proxy knows exactly, at any given time, if a message can be retrieved from cache, without having to use expensive round-trips.

It's worth noting that only the encoding side has to calculate and maintain the messages' checksums, as the decoding side only needs to save the payload. This greatly reduces the total amount of memory needed to store the 3000 and more messages that can be contained in each message store.

At first, many can think that X protocol messages present too much “variability” to be effectively cached. We have found that excluding the variant part from checksum calculation and encoding it separately, the amount of cache hits can be dramatically increased. Along the years, tuning of the encoding algorithm has permitted us to reach between 60% to 80% of cache hits on the total amount of X protocol messages encoded on the wire. For some messages, like graphic requests, images, fonts and other requests used in common office automation desktop applications, the cache hits can be 100%, allowing NX to reach compression ratios in the order of 1000:1.

We have observed that message store based encoding not only reduces the amount of bandwidth used by X protocol, but also obtains very good CPU performances with respect to any other known X compression method. In fact, given that the caching algorithm is reasonably effective, the time needed to calculate MD5 of message and send a reference to the remote peer is much lower than time needed to compress the X protocol by means of a generic algorithm, like ZLIB.

Differential X Protocol Encoding

NX proxy always tries to retrieve messages from message stores and only defaults to differential encoding if a similar message is not found. In this case, proxy encodes the message field by field, ensuring that only the significant parts are transferred and both padding bytes and implicit information are stripped out of the actual transfer.

To minimize the number of bits required to encode a message, any common request or reply has its own implementation of differential encoding. In the case of PolySegment, for example, any coordinate is sent as a signed integer representing the difference with respect to the previous X or Y coordinate. It is usually possible to encode such difference in less than 8 bits, instead of the full 16 bits required by the original X encoding. Usually each message's field has its own integer or character cache, so difference itself is matched against an integer cache. On average, a PolySegment message of 32 bytes can be fully encoded in as low as 32 bits and, in general, differential encoding allows compression ratios ranging from 5:1 to 8:1.

ZLIB Compression of Data Part

Data part of messages which don't have a specific differential encoding, is compressed through ZLIB. This compression can be fairly effective, reaching, on average, a ratio of 4:1. ZLIB compression of data part is used very seldom. Examples are black and white images, for which any other image encoding would result in worse compression, or huge GetProperty replies, for which it is not possible to determine the best differential encoding.

Storage of Data Part in Compressed Form

When data part is compressed through ZLIB, this data is stored in MessageStore in deflated form. Data is then decompressed on-the-fly at any further cache hit. This greatly helps to keep the MessageStores small, increasing the number of messages that can be cached.

Cache Based Value Encoding

NX always tries to use as few bits as possible to encode integer values. The most common fields in requests, event and replies have specific caches. There are two types of value caches, IntCaches are used to encode values of up to 32 bits, CharCaches are for values of at most 8 bits. Such caches work on a simple move-to-front algorithm. X clients are likely to use the same Window ID in multiple messages. This can result in 32 bits Window ID to be encoded in 1 bit.

Compound Cache Based Value Encoding

Let's see, now, a typical compound value cache:

```
enum T_status
{
  is_added,
  is_cached,
  is_discarded,
  is_removed
};

class StatusCache
{
  friend class EncodeBuffer;
  friend class DecodeBuffer;

  public:

  StatusCache()
  {
    slot_ = 0;
  }

  ~StatusCache()
  {
  }

  private:

  CharCache base_[4];
  unsigned char slot_;
};
```

This cache is used to tell the remote side what to do with the next X protocol message that is going to be received.

```
void encodeStatusValue(unsigned char next, StatusCache &cache)
{
    encodeCachedValue(next, 2, cache.base_[cache.slot_]);

    cache.slot_ = next;
}
```

Status is represented by, at most, 2 bits, but any new value is encoded based on the previous, in its turn based on a move-to-front CharCache. If the value is found at the first position of the CharCache associated to the current slot, it takes 1 bit to be encoded.

A similar method is used to encode GCs, Drawables and other common XIDs values that characterize the X protocol. By leveraging the recurrence of these special 32 bits values across different X protocol messages we were able to achieve very good compression ratios. ChangeGCs requests, needing on average 16/18 bytes each, are usually encoded in 16 bits, with the GC value itself encoded in 1 to 8 bits.

NX Image Compression

NX introduces a special X protocol message, NX_PutPackedImage, to deal with images. Image translation to compressed format is performed by NX X agents anytime an X_PutImage has to be sent to the X server. Agents query the proxy at session start-up to find out which image compression methods are available at the decoding side. The compression method is usually set according to the link speed. Agents compress the images and send X_PutPackedImage requests to the NX proxy that, in its turn, caches and encodes these special requests as it does for all the other X messages.

PackedImages are always kept internally by NX proxy in compressed format. They are decompressed on-the-fly, at the decoding side, whenever a X_PutImage request has to be finally sent to the X server.

NX image compression effectiveness is given by the combined effect of the compression method used by the agent (for example JPEG, PNG, RDP, TIGHT, ZLIB) and caching and differential encoding performed by the proxy. Looking at NX protocol statistics we can find that a typical JPEG image encoding can offer a ratio of 20:1. Further differential encoding and caching can offer a ratio of 100:1. The combined ratio can easily be in the order of 2000:1.

NX Image Streaming

Very good image compression methods are not enough to allow smooth user interaction in the case of narrow-band links. For example, the complete transfer of a X image having an original size of 256KB, compressed through JPEG by a factor of 20:1 to 12KB, would require 4 seconds on a modem link before any application would be able to visually respond to the outstanding user input.

NX proxy splits big messages in small chunks and streams these chunks only when no other X protocol message needs to be encoded. The algorithm penalizes X clients making extensive use of image requests and maintains the responsiveness of window managers and other concurrent clients using the X protocol in a more efficient way.

NX implements a client-server protocol between X agent and proxy to ensure that the narrow-band link between the proxy is efficiently shared between all the running applications. X agent informs the proxy when a big request is going to be sent. Agent puts the client to sleep until the request has been completely recomposed at the remote side. Agent then receives an event from proxy and can again attend to the suspended client.

The current default method to encode images in NX X sessions is JPEG, with quality determined according to link settings. Other available methods are PNG and ZLIB X bitmap compression. JPEG is a lossy image format. Most remote display systems cannot generally use JPEG as in these systems not only images but even generic screen updates, containing text or other vector graphics, are encoded as bitmaps. In these systems the use of JPEG would make the resulting output clumsy. NX, instead, preserves the original X protocol to handle text and graphics rendering and uses lossy image compression only for X requests carrying actual bitmap images.

NX Bandwidth Control

What happens if a user tries to run a screensaver in preview mode while running other X clients over the same low-bandwidth link? The screensaver will consume all the available bandwidth, leaving nothing for the other applications. In the worst case, the screensaver will fill all the available socket's TCP window, usually 64KB. Any mouse press will need something like 20 seconds before producing any visual effect.

NX uses special messages exchanged between the agent and proxy to avoid this. The agent assigns to its clients a quota of bandwidth and sends a karma message to proxy any time this quota is exceeded. The client is then put to sleep until a wakeup event is received. The proxy sends this wakeup event only if the low-bandwidth link is available to send more data, thus ensuring interactivity and fair use of the link among all the concurrent clients.

NX proxy cannot rely on its narrow-band link's TCP window being full to stop accepting data from its clients. Even reducing the TCP window to a few kilobytes would introduce too much delay between user actions and visual responses. We observed that the best results are obtained by keeping in the TCP window a constant amount of data. This amount depends on the link speed. To achieve this, NX proxy monitors the TCP window and accepts only enough bytes to make a full packet of this fixed size. In practice, the proxy always waits for small packets to be acknowledged by the remote side before accepting more data.

In the case of modem links, for example, the amount of data that proxy keeps in the TCP buffer is 2048 bytes. This guarantees a reasonable delay of 1 second between a user action and the visual response.

NX Compression of Foreign Protocols

nxdesktop and nxviewer use NX compression techniques to compress, cache and transport RDP and RFB screen updates to the client. Screen updates are usually translated in X protocol requests, for example X_PolyFillRectangle or X_ClearArea. When screen updates are encoded as images, NX transports these images in their original format, applying all the usual differential and caching algorithms. RDP or RFB images are encoded through NX_PutPackedImage requests with pack methods like PACK_RDP_COMPRESSED_256_COLORS, PACK_RFB_HEXTILE, PACK_RFB_TIGHT_PLAIN, etc. The decoding side knows how to transform these special images in X_PutImage requests.

As screen updates are transported in their native format there is almost no overhead with respect to running rdesktop or vncviewer directly on the client, with the advantage, when possible, of leveraging NX compression and the higher level X protocol primitives. nxdesktop and nxviewer can reach compression ratio ranging from 2:1 to 10:1, compared with the bandwidth usage of the same session run using the original RDP or RFB protocols.

ZLIB Stream Compression

The final stream produced by NX X protocol compression is further compressed through the generic ZLIB deflate() method. The compression level is determined according to the link settings. It goes from 9, in the case of MODEM, to 1, in the case of a link type WAN.

This final stage can reduce the amount of data traveling through the link by 30%. This is achieved without penalizing the CPU performances, as the amount of data that needs to be compressed through this generic algorithm is very small.

Persistence of Message Storse

At the time a NX session is terminated, each proxy saves the content of all its MessageStores on disk. Proxies negotiate the persistent cache at the time connection is established. If a matching cache is found, MessageStores are reloaded, greatly improving the compression performances, especially in the critical session start-up phase.

When caches are saved on disk they are checksummed by both proxies, to ensure that content is consistent with what is expected by the other peer. Checksum is validated again at the time MessageStores are reloaded. This ensures that caches are not inadvertently corrupted.

Disk Based Cache of Images

Persistence of images on disk works together with the MessageStore cache, in all cases when images cannot be found in main memory.

The algorithm is the following:

- If disk cache is enabled, the X-server side proxy saves on disk any new image bigger than the split threshold.

- Together with any new image being split (streamed), the X client side sends the message's checksum. This information is used to retrieve the object from disk.

- If the image is found on disk, the X-server side proxy can send an abort event to the X-client side.

- The abort message is asynchronous and may or may not be received before the image is completely recomposed. If the image is still being streamed, the X client side aborts the transferal and notifies the X-server side to commit the image to the MessageStore.

- At this time the X-client side agent can restart its client.

- When the X-server side proxy receives the abort message, it loads the image and transfers its content to the MessageStore.

The algorithm is based on the consideration that not all images have to be cached on disk, as the time needed to receive an abort event is often greater than the time needed to transfer the image in almost all the network conditions. The algorithm is thus carefully designed to allow both sides to work at full speed, without any of them having to resort to round-trips.

Images are saved as "I-checksum" in 16 subdirectories "I-c", where c is the first hex digit of checksum, in directory ".nx/images", in the user's home. Separation in 16 sub-directories is required to speed up the search in the case of many thousands of images present on disk.

Benchmarks show that this implementation provides very good performances. On a 28.8Kbps modem, it can almost offer the same level of perceived compression of a fully populated memory cache.