Method of Frame Buffer Transmission over Reliable Multicast Network

Choon Jin Ng and Masahiro Takatsuka
ViSLAB, The School of IT, The University of Sydney
[cjng | masa]@vislab.usyd.edu.au

Abstract

Remote desktop sharing plays an important role in remote collaboration. It acts as a "round table" where participants can perform co-operative group works and information sharing. However, many existing desktop sharing tools do not envision a large-scale collaboration environment. They mostly revolve around unicast transmission, allowing between 5-10 participating sites. However, in larger environment such as multi-site classroom, such tools cannot be used. An image (frame buffer) of a desktop needs to be individually transmitted to all sites before moving on to the next frame. This degrades the entire system performance and in some cases, limiting the contents a person can display. To circumvent this problem, multicast version of these tools were created. However, they suffer the same fate as the multicast network itself: Unreliability. This causes a chain of problems, which prohibits frame buffer to be sent efficiently over the network. This paper proposes a new solution for handling frame buffer transmission over a "reliable" multicast network.

1. Introduction

In a remote collaboration, having a What-You-See-Is-What-I-See (WYSIWIS) computing can facilitate the tracking of user's activities. This assists the exchange of workflows and ideas. Desktop sharing tool such as the Virtual Network Computing (VNC) plays a major role in enabling a WYSIWIS environment. It enables the sharing of desktop through frame buffers.

As public broadband Internet bandwidth increases, collaboration tasks have expanded from the traditional point-to-point view into one involving multi-point view. This is evident with the release of communication suites ranging from primitive point-to-point Skype, MSN Messenger, Windows Meeting Space, to group communication suites such as VSee, Conference XP and Access Grid, to web-based group communication suites such as WebEx, EVO (formerly VRVS), OpenMeetings and Adobe Connect. Among these products, one most noticeable component is the screen-sharing feature.

Typically, a screen can be shared between one-to-one party and one-to-many parties. Due to network bottleneck, the one-to-many mode usually does not scale more than 5 participants. This problem is further compounded if a many-to-many mode, which offers multiple participants to share their screen simultaneously, is to be introduced.

The root of the aforementioned scalability problem stems from the fact that most screen sharing tools are designed for unicast network operation. In a one-to-many case, the source duplicates frame buffer for all nodes. As frame buffer of a screen is fundamentally an image, distributing them continuously to all participants can consume tremendous network resources while delaying the delivery of subsequent frames due to network queues.

As such, it is becoming clear that a scalable one-to-many or many-to-many frame buffer transmission requires an alternative efficient routing scheme. Fortunately, the multicast routing scheme can replace unicast to alleviate this problem. Instead of duplicating the same packet to all participants, a multicast transmission requires a packet to be sent only once.

However, most frame buffer transmission protocols such as the one used by VNC are stream based which have presumption on states. If part of the stream is missing, then the entire "message" can be rendered invalid. Therefore, there are several inherent difficulties in adapting multicast to current "rules" of frame buffer transmission. Among them are unreliability, ordering of data and late joins

This paper introduces a different approach in handling frame buffer transmission over a multicast network. The benefit of this is to allow virtually unlimited client connection to the server, improving the performance of one-to-many mode screen sharing while enabling many-to-many mode. In addition, the worst-case cost scenario can be tied down to a constant instead of one that increases with the number of participants.

2. Related works

2.1. Virtual Network Computing (VNC)

VNC is a common remote desktop sharing tool used to control a computer remotely [4]. It consists of a server distributing screen buffers and a set of clients receiving and displaying them. Operating in "pull" architecture, clients need to explicitly send request to the server for polling screen updates.

The core protocol of VNC is the Remote Frame Buffer (RFB) protocol [5]. This protocol allows different encodings to be plugged in such as Raw, Rect, RRE, CoRRE, Tight and ZRLE. Other than Raw and Rect, the rest of the encodings typically utilizes some stateful compression schemes, rendering them unusable in unreliable network.

Moreover, the RFB does not define any late joining mechanism. Currently, all clients connecting to a VNC server will need to open an entirely new session with the server. For this reason, it takes resources on the server to keep track of each client's states. In short, for the same frame buffer update, different encoded data are sent to each client.

2.2. Compression Scheme

Many typical compression schemes such as PKZip, Zip, gzip, and ARG use dictionary compression followed by a variable-length decoder [6]. This case also holds true for *zlib*, the core compression scheme used for most VNC encodings. *Zlib* uses a combination of the LZ77 adaptive dictionary algorithm followed by the Huffman coding. It is the de facto library that implements the general purpose Deflate lossless data compression algorithm.

For the LZ77 dictionary algorithm, a list of frequently used words/data is stored in a buffer with a window size M. Any following repeated words during transmission will be represented with a symbol or dictionary index, hence shortening the data transmitted.

For Huffman coding, data statistics is collected. Overtime, these data are linked with a tree. To reach a data element, the tree is traversed using a sequence of 0s and 1s and these sequences are called the Huffman code. Hence, to compress a data, frequently used words/data is assigned a shorter Huffman code, while rarely used words/data is assigned a longer Huffman code. As such, the tree is the states in which a Huffman code refers to.

2.3. Multicast frame buffer transmission

Currently, there are already several works done to address frame buffer transmission over multicast network. These include TightProjector (TP) [1] and the Teleteaching Tool's Multicast VNC (MVNC) [2].

The fundamental mechanism of operation of these tools is based on the principle of unreliable network. For each packet, the embedded frame buffer data must be self-dependant. Hence data must be stateless and contain a complete header which describes its content. Unfortunately, this particular nature of multicast network creates several problems:

Data overheads. Since multicast operates over UDP datagram, each packet cannot be larger than the size defined by the operating system (OS). For instance, many OS defines a maximum UDP datagram size of 8192 bytes, while typical Ethernet's maximum transmission unit (MTU) is defined with the size of 1500 bytes. Furthermore, the larger a packet size, the higher chance a packet loss will occur [3]. To reduce possible network inefficiencies and to ensure greater reliability, a UDP packet is usually set to a much smaller size than a system's predefined limit. Inadvertently, this action can create data overhead since the ratio of headers to the actual amount of data stored is low.

Data compression. In the case of frame buffer transmission, the problem is further compounded when compression scheme is used. To achieve high compression ratio, it is common for compression algorithms to make references on previous received data. Instead of reduplicating data for transmission, a single symbol denoting the data can be sent instead. Therefore, there is a key trade-off between compression ratio and the length of states a compressor can track.

Data retransmission & transmission rates. Since multicast is an unreliable network, clients cannot receive all frame buffers. To compensate such error, the server usually retransmits its screen continuously even though there are no changes on the screen. The net effect is the wastage of bandwidth together with the high usage of computational power.

One possible solution includes imposing transmission rate. By carefully controlling transmission rate, a balance of CPU and bandwidth utilization can be achieved. However, with a slower retransmission rate, the receiver might have a delay in receiving frame buffer, hampering real-time application. In addition, the retransmission method is not feasible if a many-to-many mode screen sharing was to be introduced, flooding a network easily.

3. Multicast remote frame buffering architecture

3.1. Scalability of Multicast

Equation 1 describes how easily the transmission of remote frame buffers over unicast can overwhelm a network. The total bandwidth used for one server sending frame buffer b to N clients must be less than the network's capacity c. If the total bandwidth is higher than c, the network flow will be saturated with data being

queued. When such case occurs, possible solutions include dropping packets and degrading frame buffer quality.

$$N \cdot b \leq c \qquad (1)$$

In this situation, it can be easily identified that N is inversely proportional to b. With an increase in N, the maximum frame buffer size that can be carried is the bandwidth divided by N participants.

In contrast, if multicast were used, N is equal to one. In this case, the maximum frame buffer size that can be carried over the network is unaffected regardless of participant numbers. The only limitation will then be the network capacity c.

3.2. The Architecture

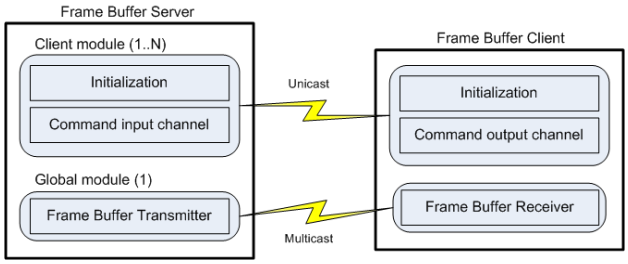

Figure 1 The architecture of the frame buffer server and the frame buffer clients.

Figure 1 illustrates the architecture for a multicast frame buffer transmission. Operating in a server-client manner, the server distributes its frame buffers to the clients. In many cases, for a client to decode incoming frame buffer, it first needs to acquire information such as the frame buffer format and its compression state. When a compression system is used in a unicast environment, the server usually create, stores and tracks each individual client's states.

Unfortunately, this technique cannot be used in a multicast environment. If individual compression state is created for each client, then for the same piece of frame buffer, the server will need to send different compressed data to the clients. This defeats the purpose of multicast transmitting the same set of data to all clients. The difference between these state managements on unicast and multicast is illustrated in Figure 2. On the other hand, sharing only one state among all clients presents a state synchronization problem. Solutions to overcome this problem are:

- *Solution 1:* Perform a state flush operation that resets the state whenever a new client enters a session.
- *Solution 2:* Transfer the states to the client.

To implement these solutions, the server requires two modules. First, a global module which broadcasts frame buffers to the clients. And second, a client module which initializes clients and notify them how to decode the frame buffer. For the former solution, if a flush operation is done too frequently, compression efficiency will be reduced. Therefore, in a multicast channel which expects many clients to join a session, this solution may be unfeasible. The trade-off between these two solutions can only be weighted by examining the characteristic of the compression algorithm used. For the latter solution, an infrastructure for state transfer was needed.

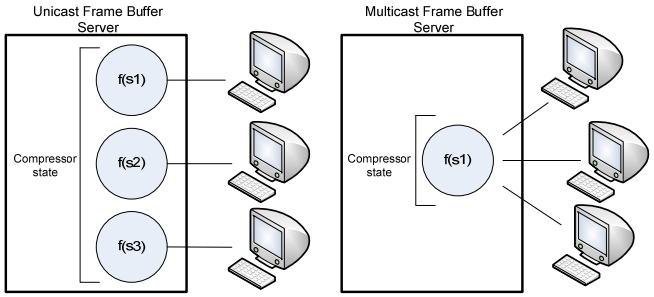

Figure 2 Using a compression function f(x), the left-hand figure shows different state are created for each client on a unicast network, resulting in different output. The right-hand figure shows state sharing with all clients on a multicast network, hence sharing the same output.

It is also possible that clients need to provide feedback to the server. For instance, clients can report the quality of services so the server can perform necessary adjustment. Functions for feedback can be embedded into the client module for individual client tracking.

For the reasons cited, this architecture requires at least two connections. For the client module, a unicast connection needs to be created for each client. Any data unique to individual are transmitted in this module. Also, the data transferred should be lightweight so as to create low overheads for every new number of clients added. For the global module, frame buffer are transmitted over multicast connection. This module multicast all frame buffer, the heavyweight data, to the clients. Depending on the frame buffer transmission format, if the data is a continuous stream, then reliability and/or ordering of data are important. In this situation, the multicast network should use a multicast protocol which has:

- ***Reliability***: Typically uses negative acknowledgement (NACK).
- ***FIFO ordering***: Each data packet has a sequence number to ensure the correct ordering of data packet transmitted.

4. The case of frame buffer transmission over multicast with RFB protocol

RFB is a stream-based protocol used by VNC. Based on the general architecture above, a new reliable multicast RFB prototype was created: *Reliable Multicast VNC (RMVNC)*.

4.1. Client Module

When a client connects to the server, a handshake process is initialized. This process allows the server to notify clients the protocol version used, security exchange, and the frame buffer format. The key information exchanged in this process is the screen size and pixel format.

For each client, the server maintains a connection with each of them to receive feedback on mouse, keyboard and RFB commands.

4.2. Global Module

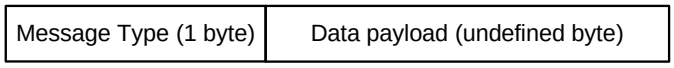

Figure 3 Server to client message format

All messages sent by the server have the protocol format listed in Figure 3. Key functions supported by the "Message Type" field include frame buffer update and "text copy and paste" operation. For each frame buffer update operation, the message must also specify the encoding used.

Currently, RFB encodings having compression capability uses *zlib* as the compressor. Following the RFC1951 Deflate specification, a compressed data set consists of a series of blocks, corresponding to successive blocks of input data [7]. Each block has a Huffman code trees describing the representation of compressed data, and a compressed data part.

In addition, the *zlib* compressor also has an additional feature not defined in the RFC 1951 specification: Flush operation. The main purpose of this feature is to ensure clients receive all scheduled data immediately instead of idling them in the buffer. The flushing modes can be categorized as: (1) Sync flush, (2) partial flush, and (3) full flush [8]. Exploiting mode (3) can allow late-join clients to keep a synchronized state. However, as stated previously, the flushing mode is not a feasible solution.

To transfer a *zlib* compressor state, the structure of the compressed block must be examined. For each compressed block, the Huffman trees are independent of previous blocks. However, the data used in the Huffman tree for all compressed blocks will refer to the same set of dictionary generated by the LZ77 algorithm. The dictionary is defined as a 32K input. Therefore, to decode all incoming data correctly, the server needs to transfer these states: (1) Dictionary and (2) Huffman trees.

However, if only the dictionary state is transferred to the clients, the inflator on the client side cannot decode the frame buffer data until a compressed block having a new Huffman trees arrives. For the case of RMVNC, this should not matter as the viewer does not need to have an initial accurate viewing of the frame buffer. In order to save bandwidth, the server initialization process can send the dictionary to the clients without the Huffman trees. In this case, the client can ignore decoding incoming compressed data until a new block with a new Huffman tree arrives.

Ideally, the state transfer mechanism should be implemented at the "Client Module" side. However, this can potentially introduce synchronization problem as retrieved state must be used at a specific point of the incoming frame buffer data in order to successfully decode it. Consequently, the state transfer was implemented on the "Global Module" to prevent the aforementioned complication. Furthermore, since the state data is about 32K in size, transferring it over multicast to all clients should not compromise performance considerably.

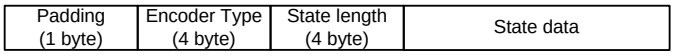

Figure 4. State transfer message format

Since the RFB does not define any protocol for transferring state, a new message type was defined for "State transfer" for the "server to client" message listed in Figure 3. The new message format is illustrated in Figure 4. It is defined such that no assumption is made on the type of compressor used. This packet is appended as a data payload for the protocol listed in Figure 3. The parameters for the state transfer message are:

Field Type	Description
Message type	4
Padding	-
Encoder Type	The encoding used.
State length	The state data's length in bytes unit.
State data	The state data.

A state transfer is required if the following conditions are true:

- A client joins a RFB session after it has started.
- The encoding scheme relies on previous data packets to decode incoming frame buffer.

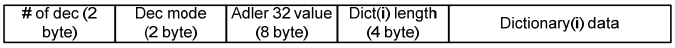

Figure 5. Z-lib state transfer message format

Since virtually all encodings which employ compression in VNC uses *zlib*, a protocol is defined to carry the *zlib* compression state. The format of this protocol is listed in Figure 5 and it is inserted as the "state data" in Figure 4. During the state restoration on the client's compressor, the Adler32 value, which is the checksum of the dictionary, needs to be replaced as well so as to prevent an "incompatible dictionary" assertion from the compressor. The decoder mode restores the stage at which the *zlib* state machine is in. With this information, it

can moves along its state machine and decode next incoming data. The definition of the parameters is:

Field Type		Description
# of dec		Numbers of decoder used.
Dec mode		*Zlib* decoding mode: METHOD, FLAG, DICT0,1,2,3,4, DONE, BAD.
Adler 32		The checksum value of the dictionary.
[i]	Dict length	The length of dictionary[i] in unit of bytes followed by its data. Note that [i] = [0 .. #dec - 1].
	Dict data	

Screen pooling

Screen pooling is also an issue with the traditional RFB pull architecture. The purpose of the pull architecture is to profile individual client connection so they can manage their own receiving rate. This is also one factor which caused RFB having difficulties in transmitting animation and videos.

However, in a multicast channel, it is no longer feasible for clients to explicitly request frame buffer updates. Firstly, if many clients are to send requests to the server continuously, the server will be overloaded with replies. Secondly, multicast dispatches data to clients all at once. Fulfilling these requests will result in data transfer rate higher than those desired by clients. In such case, the server can send:

- Frame buffer update as soon as the screen changes.
- Frame buffer update at a defined interval.

In RMVNC, the second approach was adopted for design simplicity.

To create a reliable multicast environment, RMVNC utilized the JGroups library. This library provides data transmission over multicast with NACK protocol and FIFO ordering.

5. Results

The RMVNC prototype's performance is evaluated by benchmarking it against TP and MVNC. Overall, the experiments performed showed RMVNC outperformed TP and MVNC.

During these tests, due to inability to change colour format, RMVNC and TP were transmitting at 24-bits colour while MVNC was transmitting at only 8-bits colour. In addition, the RMVNC employed Tight compression technique while the MVNC, using the Hextile encoding, achieved little compression. Though TP uses a customized compression technique, it was anticipated the compression will be inefficient due to the lack of long-term state tracking.

For all experiments, since multicast produces the same amount of data regardless of the number of clients, only 3 clients were used. Bandwidth measurement was done from the client perspective.

Figure 6 illustrate network traffics generated from the prototypes during a general desktop usage experiments. The experiment involved using a same set of mundane programs such as web browser and word processor. Since screen changes are gradual, RMVNC was able to take advantage of the advanced compression capability. As such, bandwidth usage is lower as resending images is not required. In the case of TP, even though compression is used, its efficiency is only limited to one frame. It is also worth to note that the performance of MVNC can be worse since it is only using 8-bits colour. As compression is not used in MVNC, scaling the colour to 24-bits would yield a much higher graph.

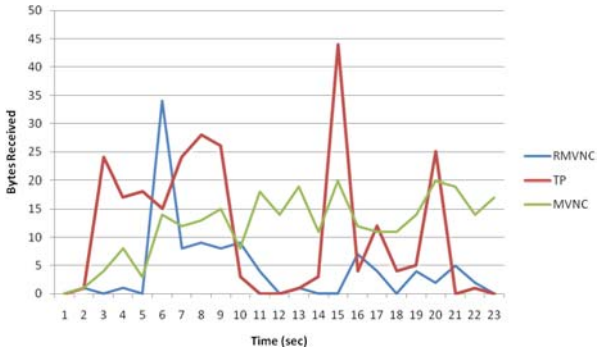

Figure 6 Desktop activities

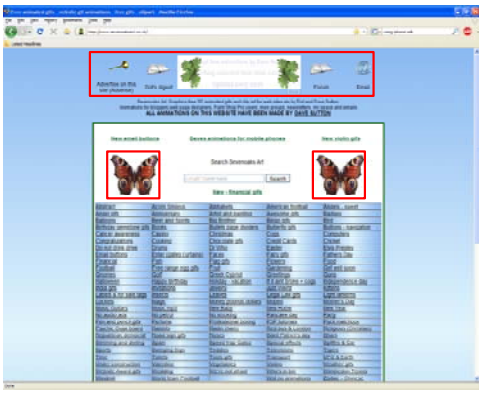

Figure 7 Static screen with little dynamic contents. The red rectangles highlight one animated banner, four animated buttons besides the banner, and two animated butterflies.

The problem is even more pronounced when the screen is static with little dynamic contents. The graph in Figure 8 illustrates the bandwidth consumed for this condition. The experiment involved transmitting the screen showed in Figure 7. The red highlight shows the part of animation with repetitive patterns, allowing compression efficiency to be tested. At that time, no user activities were present except for the on-screen animation. The behaviour on when to transmit frame buffer differs on RMVNC, TP and MVNC. TP and RMVNC transmit

frame buffer only when the screen changes. From the graph, MVNC was transmitting at a continuous stream of 8-bits colour to reduce chances of data loss. With the absent of reliable data channel, MVNC cannot verify if clients had received the frame buffers. As such, it had to constantly resend the entire screen in sequence. Again, if MVNC is switched to 24-bits, its graph would be higher than that of TP and RMVNC. As for TP, the graph spikes indicate the animation was sent without efficient compression. The compression scheme is not capable of "remembering" the previous colour patterns that had been used. On the other hand, RMVNC had a much better performance. With a low steady stream, RMVNC had better reflection on changed data.

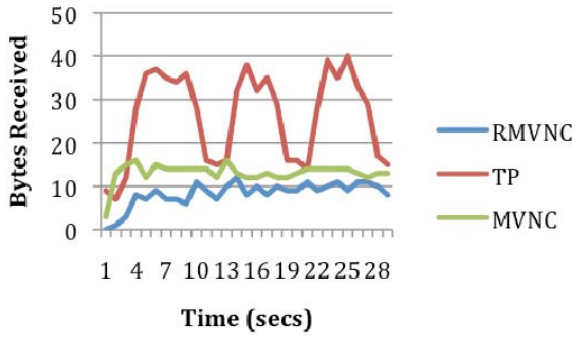

Figure 8 Static page with some dynamic content

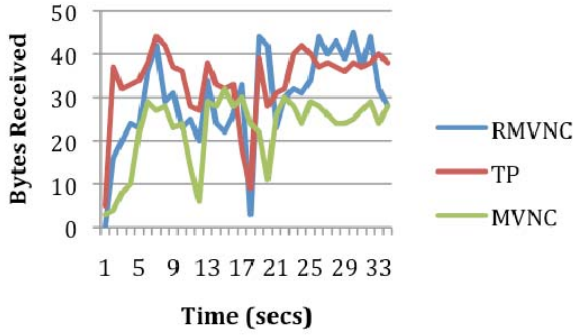

Figure 9 YouTube video

In Figure 9, intense screen activity was tested. A standard size YouTube video was replayed. All prototypes had similar bandwidth usage with MVNC being the lowest, albeit with 8-bits colour. This suggested the nature of a highly variable data environment can reduce the efficiency of the compression method. However, compared to the other prototypes, RMVNC was observed having a smoother video replay. Such can be attributed to the long-term "memory" compression in the RMVNC, enabling the use of more efficient encodings.

6. Conclusion

Previously in MVNC, the effect of packet lost was taken into consideration. To prevent that from happening, the entire screen buffer had to be constantly retransmitted in a partially sequential manner, increasing network utilization. For such reason, the MVNC performs relatively slow on screen updates. Owing to this fact, MVNC was limited to 8-bits colour.

With reliable multicast technology introduced into the RFB protocol, frame buffer no longer floods a network. If packet lost occurred, the client requests the server for retransmission, hence reducing network utilization.

An interesting part of the result is that TP had fared better than MVNC even though no reliable multicast was used. This happens as TP has no concern on reliability and so does not perform retransmissions. In a small LAN, TP can be unsusceptible to packet loses. However, once placed in a WAN, it is very likely suffers packet loss and clients will have incomplete screen rendering.

Moreover, having state transfer capability allowed RMVNC to use a wide range of compression methods. With this, RMVNC can save tremendous bandwidth on screen that has little changes, such as word processing.

Finally, unlike other prototypes listed here, the architecture introduced does not focus only on the broadcasting aspect. It also introduced 2-ways communication, allowing clients to interactively control the server's screen.

The paper contributed a method to transfer frame buffer over a reliable multicast network. First, an architecture is devised. Then, a prototype is created based on the architecture. The benefits of this architecture are:

- Lower network bandwidth consumption.
- Allows the use of compression technique with state transfer technology. This directly allows higher quality transmission.
- 2-way multicast interactive control.

7. References

[1] V. Vologzhanin, "TightProjector Software." vol. 2008, 2007.

[2] P. Ziewer and P. D. H. Seidl, "Transparent TeleTeaching," in *In Proceedings of ASCILITE 2002*, Auckland, NZ., 2002.

[3] H. Sawashima, Y. Hori, and H. Sunahara, "Characteristics of UDP Packet Loss: Effect of TCP Traffic," in *Proceedings of INET '97: The Seventh Annual Conference of the Internet Society*, Kuala Lumpur, Malaysia, 1997.

[4] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *Internet Computing, IEEE*, vol. 2, pp. 33-38, 1998.

[5] T. Richardson, "The RFB Protocol," 2007.

[6] K. Sayood, *Introduction to Data Compression*, 2nd edition ed.: Morgan Kaufmann Publishers, 2000.

[7] L. P. Deutsch, "RFC1951: DEFLATE Compressed Data Format Specification version," Network Working Group 1996.

[8] T. Pornin, "Zlib Flush Modes." vol. 2008 Québec, 2007.