Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp809-814)

A Software-based Real-time Video Broadcasting System

MING-CHUN CHENG, SHYAN-MING YUAN
Dept. of Computer & Information Science
National Chiao Tung University
1001 Ta Hsueh Road, Hsinchu, Taiwan 300
TAIWAN, REPUBLIC OF CHINA

Abstract: - There are currently many video broadcasting products and applications, such as projectors, learning systems, and video streaming systems, but they are either hardware implementations or non real-time implementations. Additionally, almost all of them do not support a one-to-many model. To solve above problems, a novel software-based approach capable of rendering full screen 20 frames per second under 800x600x16 resolution to one or two more computers is proposed in this paper. Different methods are investigated and the most suitable one chosen to achieve this goal. In addition, this system is currently applied to one-to-many video learning systems.

Key-Words: - video broadcasting, screen capture, screen changes detection, one-to-many multicast

1 Introduction

Video is a good communication tool, and a video broadcasting system can deliver some screen contents to some computers to show. It can be applied to many different fields, such as education, and entertainment, as well as business. A great deal of effort has been made today on many video broadcasting products and applications, such as projectors, video learning systems, and video streaming systems. However, they are either hardware-based implementations [1][2] or non real-time implementations [3]. The term real-time in this paper means all screen contents have to be synchronized and no frame delay occurs. In other words, buffering technology [12], which is widely used by streaming system, can not be exploited in the system. Besides, Users have to buy certain equipment or spend time preparing for content before using them. In addition, almost of them [4][5][6] do not well support a one-to-many model.

Developing a software-based real-time video broadcasting system faces many problems. Performance is the main problem, because video data is too large to process or transmit well. Designing and implementing a software-based real-time video broadcasting system capable of rendering full screen 20 frames per second under 800x600x16 resolution to one or two more computers is the aim of this paper. Many different methods are analyzed and the most suitable chosen to achieve this goal.

This paper is organized as follows. Section 2 introduces display system fundamentals and Section 3 explains design and implementation. Section 4 describes applications. Finally, Section 5 presents the conclusions.

2 Fundamentals of Display System

An inquiry of display system fundamentals must first be made. The hardware and software display system architectures are introduced in this section. Because the current system is designed and implemented for the Microsoft Windows platform, the following content focuses on Microsoft Windows.

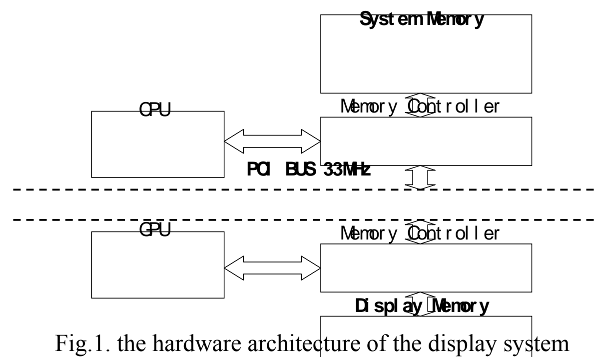


Fig.1. the hardware architecture of the display system

2.1 Hardware Architecture

There are two separate memories in the display system as Fig.1 shows. One is system memory and the other is display memory. The data generally moves between both. Such moves are handled by the CPU and have to go through the performance-limiting PCI bus, with a bandwidth of about

33MBytes/s (33MHz * 8 Bytes). To solve this problem, AGP was proposed. AGP has more bandwidth than the PCI bus, and the display system can move data more efficiently by going through the AGP. Nevertheless, this hardware acceleration is only for one direction, from system memory to display memory. As a result, AGP does not help screen capture, which has to move data from display memory to system memory.

2.2 Software Architecture

Fig.2 depicts the display system software architecture, and this architecture will be explained by an example. When an application wants to draw something on the screen, it will call Win32 GDI functions first, and these Win32 GDI functions will then call the graphics engine from user mode to kernel mode. The graphics engine will also mediate these requests to the corresponding graphics driver, which is responsible for rendering. Finally, the graphics driver translates these requests into commands for the video hardware to draw graphics on the screen.

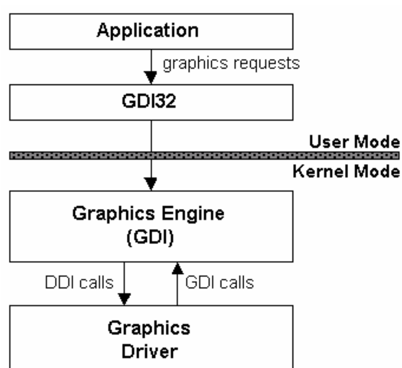


Fig.2. the software architecture of the display system

3 Design and Implementation

There are two roles in the system, a sender and one or two more receivers. The sender is responsible to capture its screen, encode the captured data, and send them to receivers. A receiver is responsible to receive the data from the sender and update its screen. The details of these actions are illustrated as Fig.3.

In Fig.3, the left part represents sender and the right part represents receiver. The arrow indicates the processing direction. On the sender side, there are five steps to process from top to bottom, and each of them will be explained in different sub-sections. The first step is to detect which areas on screen had been changed from the last detection, and the second step is to merge the results of the first step. The third step is to capture these screen areas described by the results of the second step. The fourth step is to compress and encode them into update commands, and the last step is to send them out via data channel. Processing from step 1 to step 5 is called a round. Moreover, the system has to process a round at least every 50 ms to generate 20 frames per second, otherwise users will sense frame delay. On the receiver side, when receiving update commands from sender, the first step is to decode and decompress them, and then update its screen according to commands received in the first step.

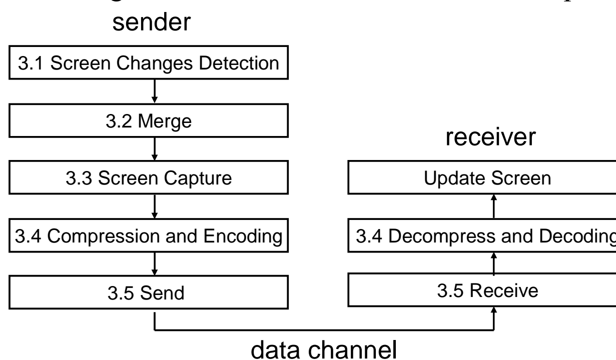


Fig.3. the flow chart of processing steps

Because of the real-time characteristic, every step should be as fast as possible. However, the fastest algorithm for one of the steps may not be the most suitable for the overall system. To take a simple example, in the compression and encoding step, the faster compression algorithm may have less compression ratio, causing the next step to spend more time on sending them. The authors in this section will discuss many different mechanisms or algorithms, analyze the trade-off among them, and choose the most suitable one in a different step.

3.1 Screen Changes Detection

Because of insufficient PCI bus bandwidth, introduced in section 2.1, detecting screen changes is an important consideration. The purpose of screen change detection is to reduce data size in order to improve system performance. Based on section 2.2, there are many opportunities to intercept the drawing requests from application. According to the parameters of these requests, the system can know what happens on screen. Table 1 shows four methods to detect screen changes and the differences among them. The authors use five criteria to compare them. The first criterion is to see whether or not to reboot during first installation. If rebooting is necessary, it may make for a bad end-user experience. The second criterion is to judge development difficulties. More development difficulties imply more side effects. The third

criterion is to check whether or not to disable DirectDraw when activated. If DirectDraw does not disable, something on the screen may not capture by these methods. The fourth criterion indicates which versions of Microsoft Windows are supported. The last criterion is to see whether or not to reboot when unloading these hooks or drivers.

	DDIhook	Graphics Driver hook	GDI32 hook	Mirror Driver
Reboot when first installation?	No	Yes	Yes	No
Difficulties in implementation	Easy	Tedious work	Tedious work	Easy
Disable DirectDraw when activated?	No	No	No	Yes
OS requirement	Win95/98	Windows	Windows	Win2000
Reboot when de-activated?	Yes	Yes	Yes	No

Table 1. the characteristics of different screen change detection mechanisms

☒ **DDI Hook** [7]

This method exploits an undocumented API, SetDDIHook, provided by Microsoft. This API can intercept all DDI functions, but it only supports Windows 95/98 and it will consume more CPU resource than others methods.

☒ **Graphics Driver Hook [7]**

This method exploits wrapper technology. It replaces the original graphics driver, which is a DLL file, by a wrapper driver, which is a DLL file also. All DDI calls will first be intercepted by this wrapper driver and then passed to the original driver. This method has to implement all DDI calls that the original DLL supports, and it is a tedious work.

☒ **GDI32 Hook [7]**

This method is almost the same as the graphics driver hook mentioned above, except it is in user mode, and this method replaces GDI32 DLL instead of graphics driver.

☒ **Mirror Driver [7]**

Mirror driver is provided by Windows 2000 and later. A mirror driver is a display driver for a virtual device that mirrors the drawing operations of one or more additional physical display devices. After it is activated, when the system draws to the primary video device at a location inside the mirrored area, a copy of the drawing operation is executed on the mirrored video device in real time. The system can track the screen update using a copy of the drawing operations.

After analyzing the above four methods, the authors choose mirror driver to detect screen changes. In addition, every screen change detected by the above mechanisms is described by a rectangular area, which is denoted by two points, such as (x1, y1, x2, y2). An example is shown as Fig.4. The outer frame represents an entire screen, and the shading portion represents changes in this area.

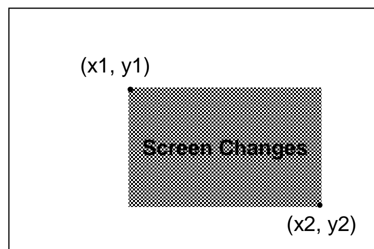
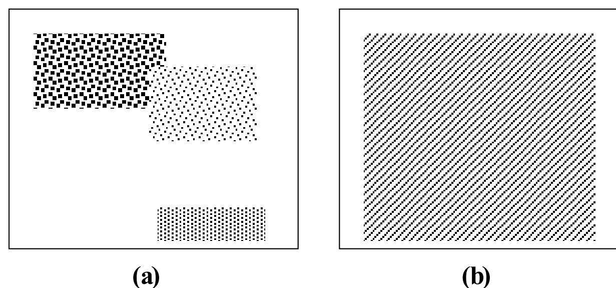


Fig.4. this figure represents an entire screen, and shading portion represents a screen change.

3.2 Merging

There may be more than 100 screen changes every 50 ms, and each can be represented by a rectangular area as Fig.4 shows. If capturing each of them individually, it is too many capture actions to complete them in time. To reduce the number of capture actions, the system uses a merging algorithm to merge related areas into a larger one. The algorithm is described below.

The simplest merging algorithm is to find the top-left and bottom-right points among all screen change areas. Consequently, these two points represent a larger area covering all previous screen change areas, and the system can handle it instead of every individual area. On the other hand, this algorithm is not concise enough; it may waste time handling too many un-changed areas as Fig.5 shows. Three screen changes are detected before merging in Fig.5. After merging, these three areas are covered by another larger area, which covers, however, too many un-changed areas.



(a) **(b)**

Fig.5. (a) before merging; there are three screen changes, and these changes are denoted by shading color. (b) after merging; the shading portion represents the merging result

In order to solve this problem, the system first divides the screen into several rectangular blocks as Fig.6 demonstrates. Every block uses the same algorithm presented above, and this step is called Merging Phase 1. After Merging Phase 1, the

system will merge areas in neighbour blocks, called Merging Phase 2. This algorithm is not only more concise, but also limits the number of change areas to handle. The upper bound is the half number of total blocks. An example of this algorithm is demonstrated as Fig.6.

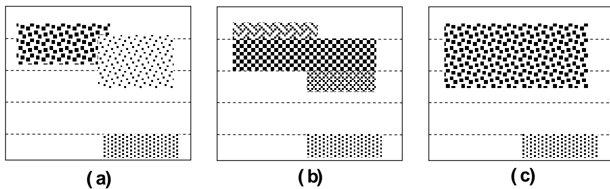


Fig.6. (a) Merging Phase 1. (b) Merging Phase 2. (c) After merging

In Fig.6, after Merging Phase 1, there are four areas left. Moreover, above three areas are neighbours, and they will be merged after Merging Phase 2.

3.3 Screen Capture

In order to prevent users from sensing frame delay, frame rate has to be larger than 20 fps. Thus, the system has to choose a screen capture mechanism capable of capturing full screen at least 20 times per second.

Fig.7 shows the performance of three different mechanisms to capture screen. This section describes them individually below.

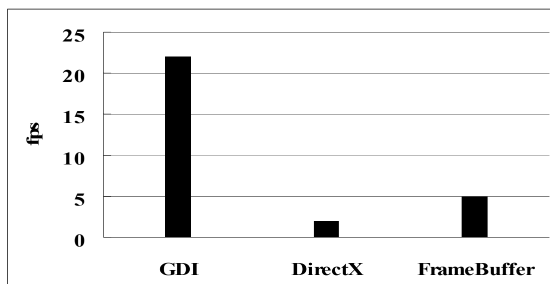


Fig.7. the performance of different screen capture mechanisms

☒ **GDI** [8]

This method uses GDI API, BitBlt, to capture screen. The BitBlt function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context. By this method, the system can get a desired format of byte buffer.

☒ **DirectX** [8]

DirectX is a set of multimedia APIs that Microsoft provides. Every DirectX application contains what we call buffer or surface to hold the video memory contents related to that application. These buffers are called the back buffer of the application. There is also another buffer that every application can access called front buffer.

The front buffer holds video memory related to the desktop contents. By accessing the front buffer from the DirectX application, the screen contents at that moment can be captured.

☒ **FrameBuffer** [7]

A frame buffer is the dedicated memory on a graphics adapter. It is possible to write a driver to access FrameBuffer directly. Nevertheless, a different graphics adapter may have a different format from FrameBuffer, and direct access performance is not good, as explained in section 2.1.

After conducting experiments on the above three methods, the authors choose GDI to capture screen. In addition, whatever the sender color depth and format, all captured data are converted into the R5G6B5 format, and the system leverages MMX instructions to speed-up transform speed.

3.4 Compression and Encoding

To reduce bandwidth usage, the system compresses the data before sending. Based on test results from compression.ca [9], the authors choose a sufficiently fast and good compression ratio algorithm called LZOP to compress data.

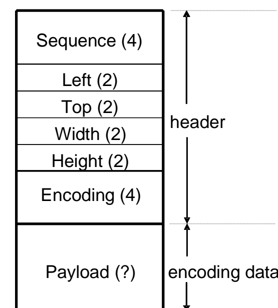


Fig.8. the encoding format

After compressing, the system has to encapsulate the compressed data into a specific format, called update command, and this step is called encoding. The encoding format is shown as Fig.8.

This encoding format is almost the same as RFB (Remote Frame Buffer), used by VNC. An RFB packet can be seen as having two parts, header and data. The header includes the update command sequence number, the area position information, and compressing method used. Moreover, the data part contains compressed data. When receiving an RFB packet (update command), the receiver can decompress the data part using the method the header part declares, and then copy the decompressed data into FrameBuffer according to the area position information.

3.5 Transmission

The system exploits a modified UDP protocol to send screen update commands from sender to receivers for two reasons. The first is that the screen update command is time-critical, and some lost commands may be out-of-date as well as the system does not need to re-transmit these lost commands. In other words, TCP is not suitable for a wireless environment [10]. The second reason is that TCP is not suitable for a one-to-many scenario [11]. TCP needs more bandwidth than UDP when the sender transmits the same data to two more receivers.

UDP, however, is not good enough for transmitting screen update commands. To introduce easily, the screen is divided into only four blocks, and a block within a shading color indicates that the block has been changed since the last time. Every figure has two parts, the upper part represents sender, and the lower part represents receiver. The arrows represent screen changes time by time. Furthermore, α, β, and γ represent update commands, and alphabets, A, B, C and so on, represent block show content.

When some of the blocks change in the sender-side, after processing, the sender will send these update commands to receivers. When a receiver receives update commands, it will refresh its screen according to these commands as soon as possible.

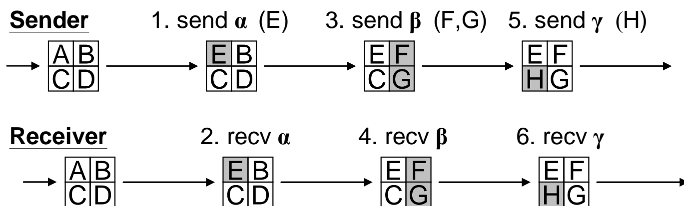


Fig.9. normal situation

Fig.9 expresses a normal situation (no update command lost), and the details are explained as follows:

1. The sender detects an upper-left block change, the content of which had been changed from A to E, encodes this information into α, and sends it out.
2. The receiver receives α, decodes α, and updates its screen according to α. After processing, the upper-left block content is changed from A to E.
3. The sender detects a change in the right two blocks, encodes this information into β, and sends it out.
4. The receiver receives β, decodes β, and updates its screen.
5. The sender detects a change in the lower-left block, encodes this information into γ, and sends it out.
6. The receiver receives γ, decodes γ, and updates its screen.

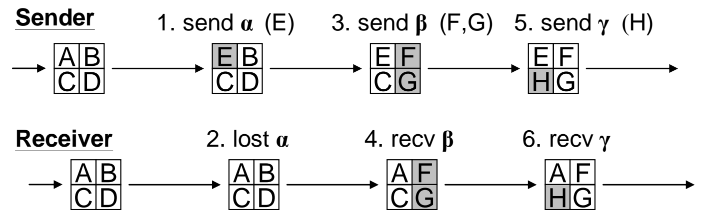


Fig.10. error situation when using UDP

Fig.10 illustrates an error situation (one command lost). Because of lost packets, it makes some receiver screen blocks inconsistent. For example, in Fig.10, the upper-left block of the last screen in the receiver should be E, but is A, caused by the loss of α.

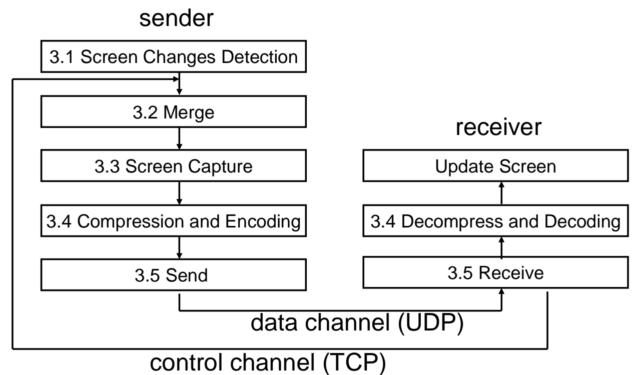


Fig.11. the revised flow chart of processing steps

To overcome the above problem, the authors propose a modified UDP, named MDP. The MDP concept is that a receiver can tell the sender which update commands the receiver does not receive via the control channel, which is a TCP connection. Moreover, the sender can merge the screen area covered by these lost update commands into the next update command instead of re-transmitting the original update command as Fig.11 shows.

To implement MDP, the sender has to record every update command position, including left, top, width, and height, and every record expires after 2 seconds. Moreover, every update command from sender has a unique sequence number to detect lost commands.

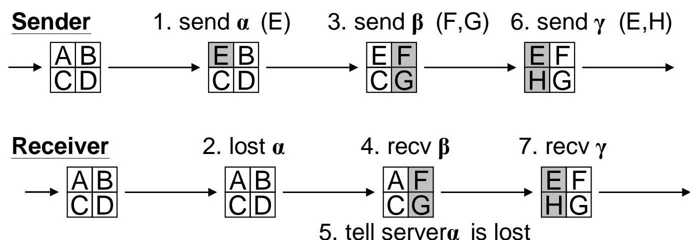


Fig.12. error situation when using MDP

Fig.12 is an example explaining MDP. Steps 1, 2, 3, 4 are the same as Fig.10. After receiving β, the receiver will discover some lost commands by comparing sequence numbers. Then, the receiver tells the sender α is lost. When the sender receives lost information from the receiver, it will query previous records to see what screen areas these lost commands cover. Finally, the sender adds this area into merge phase (section 3.2), and these areas are considered as changed areas.

4 Applications

There are many commercial products for the video learning system, but most of them are hardware-based.

In this application, there are two roles in the system, teacher (sender) and student (receiver). When the system is activated, the content of the students' screen will be the same as the teacher's screen. Thus, this system can be used for teaching, demo, and so on. The authors apply the proposed system to a video learning system to reduce cost, because all necessary equipment is existence in traditional computer rooms. Fig.13 illustrates the architecture of the software-based system, and all equipment is interconnected with a wired network, for example, 100 Mbps Ethernet.

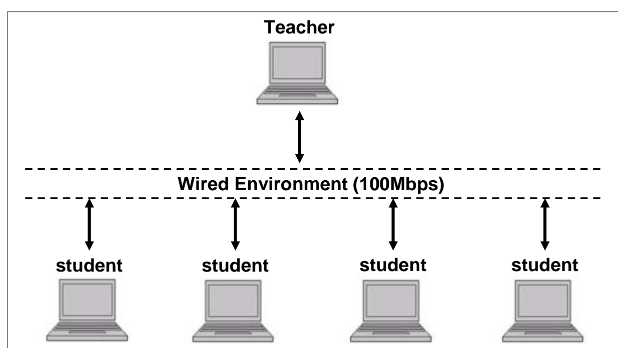


Fig.13. the architecture of the video learning system

5 Conclusions

There are many issues which need to be addressed when designing and implementing a software-based real-time video broadcasting system. The authors survey many different methods and mechanisms for each processing step, in this paper, and explain how to choose the most suitable ones. A novel transmission mechanism is additionally proposed in this paper to support a one-to-many model. Currently, this system is applied video learning systems. Reducing bandwidth usage will be the authors' future focus.

References:

[1] D-Link DPG-2000W, *http://support.dlink.com/products/view.asp?productid=DPG%2D2000W*

[2] NEC's Wireless MT1065, *http://www.projectorcentral.com/wireless_nec_mt1065.htm*

[3] The Windows Media Technology Web Page, *http://www.microsoft.com/windows/windowsmedia.*

[4] VNC, *http://www.realvnc.com*

[5] Ricardo A. Baratto, Jason Nieh, and Leo Kim, "THINC: A Remote Display Architecture for Thin-Client Computing", *Technical Report CUCS-027-04*, Department of Computer Science, Columbia University, July 2004.

[6] S. Jae Yang, Jason Nieh, Matt Selsky, and Nikhil Tiwari, "The Performance of Remote Display Mechanism for Thin-Client Computing", *Proc. of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 10-15, 2002, pp. 131-146

[7] Microsoft Windows DDK Document

[8] Microsoft MSDN library

[9] Archive Comparison Test, *http://compression.ca/act/act-summary.html*

[10] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links", *IEEE/ACM Transactions on Networks*, 1997.

[11] Tsun-Yu Hsiao, Ming-Chun Cheng, Hsin-Ta Chiao, Shyan-Ming Yuan, "FJM: A High Performance Java Message Library", *IEEE International Conference on Cluster Computing 2003*, Hong Kong, Dec 1-4, 2003, pp.460-463