A Hybrid Thin-Client protocol for Multimedia Streaming and Interactive Gaming Applications

D. De Winter
davy.dewinter@hogent.be

P. Simoens
psimoens@intec.ugent.be

L. Deboosere
ldeboosere@intec.ugent.be

F. De Turck, J. Moreau, B. Dhoedt, P. Demeester

Ghent University - Hogeschool Gent INWE - IBBT - IMEC, Department of Information Technology
Gaston Crommenlaan 8 bus 201, 9050 Gent, Belgium
Tel: +3293314900, Fax: +3293314899

ABSTRACT

Despite the growing popularity and advantages of thin-client systems, they still have some important shortcomings. Current thin-client systems are ideally suited to be used with classic office-applications but as soon as multimedia and 3D gaming applications are used they require a large amount of bandwidth and processing power. Furthermore, most of these applications heavily rely on the Graphical Processing Unit (GPU). Due to the architectural design of thin-client systems, they cannot profit from the GPU resulting in slow performance and bad image quality. In this paper, we propose a thin-client system which addresses these problems: we introduce a realtime desktopstreamer using a videocodec to stream the graphical output of applications after GPU-processing to a thin-client device, capable of decoding a videostream. We compare this approach to a number of popular classic thin-client systems in terms of bandwidth, delay and image quality. The outcome is an architecture for a hybrid protocol, which can dynamically switch between a classic thin-client protocol and realtime desktopstreaming.

1. INTRODUCTION

The centralized computer model has evolved to a more distributed model where almost every home or office user has his own PC, mostly connected to a network. Such a decentralized model has a lot of problems: due to the lack of knowledge, users in a home-environment are vulnerable to security-related attacks, to data-losses etc. Also in corporate environments, costs of maintaining the infrastructure can become very high. Current trends show an new evolution to a centralized model; examples are the new emerging web-services of Google and Windows Live of Microsoft (both using AJAX (acronym for Asynchronous JavaScript And XML) to build web-services with a look-and-feel of normal applications). Portable devices can improve user mobility but as soon as resource intensive applications are used, battery lifetime will be too limited. Important data can also be lost due to theft or laptop damage. Thin-client systems can offer an all-round solution to tackle these problems. A typical thin-client system consists of a client and a server communicating with each other using a remote display protocol over a network connection. The application is executed on the server, and only the graphical output is sent to the client. Classic thin-client systems however suffer from a number of problems: they cannot be used for applications relying heavily on GPU-acceleration or having a lot of non-related screen updates. The remainder of this paper is structured as follows. After a short overview of related work, a high-level architecture for a hybrid thin-client protocol at the server is introduced. Furthermore, we will describe the implementation of our realtime desktop streamer. We will determine the optimal parameter settings of the used videocodec. Next, a comparison of classic thin-client protocols with realtime desktopstreaming is given, motivating our choice for a hybrid thin-client protocol. Finally, a conclusion is drawn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NOSSDAV '06 Newport, Rhode Island USA
Copyright 2006 ACM 1-59593-285-2/06/0005 ...$5.00.

2. RELATED WORK

Figure 1 gives an overview of commonly used thin-client systems. An evaluation has been given in [2] and [5]. Graphical output from the applications can be sent at different layers: the highest layer is a high-level graphical library such as the Gtk-toolkit (3). Commands to draw comboboxes, buttons, etc. are sent to the thin-client. An alternative is to use a low-level graphics library such as the X-Window protocol. The high-level graphics library (3) translates the complex commands to simpler ones understood by the X-Window protocol (2). These commands are sent to the X-server over a network (4), which translates the commands to low-level commands the graphic driver understands. To support remote X-sessions over low bandwidth connections, powerful optimisations such as the NX-protocol have been proposed: an nxagent (5) and an nxproxy (6) are placed between the Xlib-library and the X-server to cache and compress the commands. THiNC [2] only sends low-level videocard driver(7) commands over the network to the thin-client (8). The RFB protocol also uses a pseudo videocard driver (9) invoked by the X-server and writes the graphical output to a software-framebuffer. The content of this framebuffer

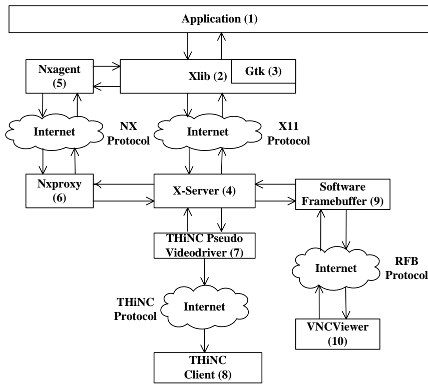

Figure 1: Components of classic thin-clients

is sent to the thin-client.
All current thin-client systems have two common problems: first, the graphic commands are intercepted at a software-level before GPU-processing takes place. An example is 3D-applications using an OpenGL API. The OpenGL-libraries can interact directly with the video-card driver. Complex calculations are performed by the GPU and the result is directly written to a hardware framebuffer not accessible by the thin-client system. THiNC offers a solution to this problem by implementing in the virtual video driver also the 3D-commands called by e.g. OpenGL, and sending them to the thin-client. The thin-client must then perform the GPU calculations, and we can no longer call it a real thin-client. If a thin-client has its own CPU, GPU, memory, etc., a lot of the advantages of "real" thin-clients are lost: the end-user has to upgrade and maintain his system again. Second, classic thin-client protocols are ideally suited for programs producing low-motion graphical output, but not for e.g. video. We have already mentioned the use of web-services to replace normal desktop-applications. The main drawback there is that applications have to be rewritten, and there is no support for GPU-intensive applications. A first suggestion to use video streaming to encode the graphical output of an application and send it that way to a thin-client has been proposed in [3]. This paper describes a hybrid protocol which implements this solution efficiently.

3. REALTIME DESKTOP-STREAMING

3.1 A hybrid architecture

We propose a hybrid thin-client protocol using a combination of a classic thin-client protocol and videostreaming to send the graphical output of an application to a thin-client device. Figure 2 gives a schematic overview of such an architecture at the server. An extra *driver decision* abstraction layer is inserted between the graphical libraries and the device driver layer. This layer will decide if commands coming for the graphical libraries are directly passed to a native videocard device driver, or a pseudo device driver implementing e.g. the VNC RFB-protocol. If an application calls functions from a library requiring GPU support, the abstraction layer can inspect the complexity of the commands and pass them to the native device driver or the pseudo device driver (and let them process by the GPU resp. CPU). If they are passed to the native device driver, frames will

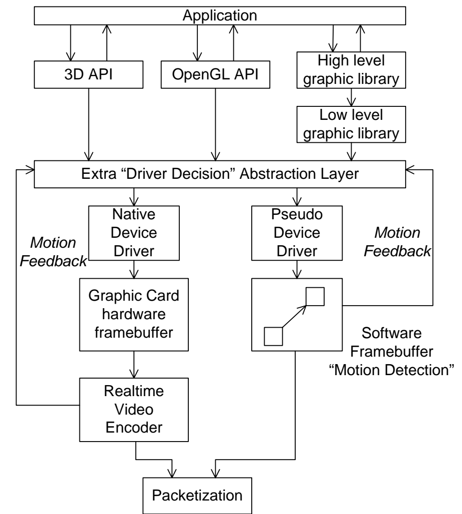

Figure 2: Architecture of a hybrid thin-client protocol at the server

be encoded by a realtime video encoder after GPU processing and streamed to the thin-client device. The decision to which driver the calls will be passed, can also be based on the amount of motion in the images. If the amount of motion in the software framebuffer exceeds a particular treshold, the abstraction layer will decide to use the native device driver and stream the output. The realtime video encoder should also give *motion feedback* to the abstraction layer. If the amount of motion is below a certain treshold, the pseudo device driver will be used again. Instead of a VNC-like pseudo device driver, other systems can be used (e.g. by sending the low level graphic commands directly to the thin-client after compression in an NX-way). A switch should happen transparant for the end-user (e.g. by using the latest frame of the video-encoder as the first frame for the software framebuffer and vice versa). Applications must not be modified to be used in this thin-client architecture. The actual protocol will use a combination of e.g. the RTP / RTCP protocol family to send the videostream and a classic thin-client protocol such as FreeNX if no streaming is required. Furthermore, the protocol will also need some enhancements to send a e.g. a trigger to the thin client if or before an actual switch happens.

3.2 Minimal buffering

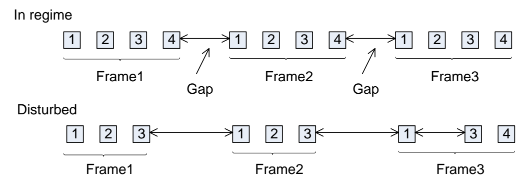

Figure 3: Packets of the videostream as they arrive at the thin-client

To guarantee decent user responsiveness, the interaction delay (the time between clicking a button and seeing the result of that click-operation) must be kept below 80ms. If we

use videostreaming to send the graphical output of an application buffering at thin-client side must be avoided. If there is enough bandwidth, the thin-client sees the *in regime* scenario (figure 3): per frame, a burst of packets arrives in the videostream, with gaps between 2 frames (the time needed to read out a new frame and encoding it). The *in regime* scenario is optimal, minimizing interaction delay. The thin-client can read out the packets for exactly one frame, and during the gap, it has enough time to decode and display the frame. In that way, no buffering is needed. When the network runs out of capacity, the *disturbed* scenario will be seen: due to buffering and/or shaping at routers, gaps can be larger or smaller and bursts can contain packets of different frames. Packets can also be dropped. If the *disturbed* scenario is observed by the thin-client, a feedback message is sent to the streamer to reduce bitrate by lowering quality or resolution of the videostream until packets arrive back *in regime*. If there is more than 1 frame in the buffer, only the most recent frame should be decoded and the others discarded (the principle of circular buffer).

3.3 Implementation details

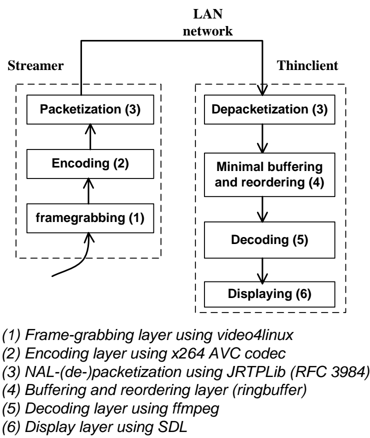

Figure 4: The different software components of our desktop streamer

We developed a realtime desktop streamer for our tests to compare desktop streaming with classic thin-client protocols. The desktop streamer streams the graphical output of an application after GPU-processing via a separate videostreamer to a thin-client. Figure 4 shows the high-level components of our software: at the streamer, frames (coming from the server after GPU processing) are read out of a frame-grabber, using video for linux. The frames are in realtime encoded by the x264 [1] H264/AVC videocodec. The encoded data is packetized in Network Abstraction Layer (NAL) units. The large NAL-units are split up in smaller fragmentation units (FUs), to reduce image-quality degradation in lossy networks. The FUs are sent over the network in RTP-packets. The RTP-packetization, session setup and RTCP feedback channels are handled by the JRTP-lib [4]. The encoder is implemented as one loop, sending out an encoded frame every 40 ms. User interaction is guaranteed by the XTest extension library of the X-server. If a user clicks a particular point at the videoscreen on his thin-client, coordinates are translated to the X-server (the same goes for other events such as keyboard-events). To minimize interaction delay, the buffering , decoding , and display-processes at the thin-client side are also implemented as one loop (except one extra thread for tracking user-actions). A buffering algorithm to detect the *disturbed* scenario (see previous section) is also implemented. For our tests, packets always arrive *in regime* (on the high speed LAN network).

4. PERFORMANCE EVALUATION

4.1 Test scenarios

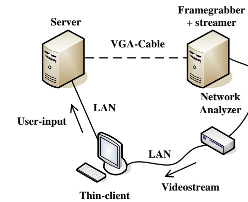

Figure 5: Testbed for streaming desktop evaluation

Figure 5 shows our testbed. The testmachines are PCs connected by a 100 Mbit LAN-network. All machines have an AMD Athlon64 3500+ CPU with 512 MByte of memory and run Gentoo Linux with a 2.6.14 kernel. The VGA-cable of the server is connected to a framegrabber capable of capturing 25 frames per second. These frames are in realtime encoded and sent over a network analyzer to another PC functioning as thin-client and capable of decoding and displaying the videostream. We use four distinguished scenarios as indicated in Table 1. The gaming scenario could only be tested with realtime desktopstreaming as such an application depends on the GPU (due to the use of OpenGL). To determine the optimal videocodec settings for streamingdesktop, we have prerecorded a frame-sequence (25 fps) for each scenario on harddisk, instead of reading directly from the framegrabber. In that way we have identical frames for each test. All applications, except the video-sequence (640×480) run at a resolution of 1024×768, but are scaled down by the framegrabber to 640×480 (the technical limit).

4.2 Desktop streaming evaluation

We evaluate the performance of our desktop streamer and determine the optimal codec settings in terms of bandwidth and delay (also implying processingoverhead). This is not a classic evaluation of a videocodec: we are especially interested in the suitability of such a codec for desktopstreaming and the influence of the different parameters on delay and bandwidth (both must be minimal). We only mention the most influencing parameters. For each test, we have varied one parameter while keeping the others constant. All tested parameters have almost no influence on image quality (formally confirmed by the PSNR-ratio). For the reference settings of the codec, we do not use B-frames or extra reference frames, a variable GOP-size between 25 and 250, a diamond motion estimation search algorithm and a motion estimation range of 16 pixels. Furthermore, CABAC and subpixel motion estimation are disabled, and no direct motion vectors are used.
The total delay between reading a frame at the server and displaying at thin-client side for the most influencing parameters is shown in figure 6 for the gaming-scenario. Delay-trends are similar for the other scenarios, but the overall

Table 1: Overview of our 4 test scenarios

Scenario	*Description*
Office	A sequence of actions in **openoffice** (e.g. cut and paste of images, typing, scrolling, creating tables, etc.) recorded with and played back with Xnee. **time: 180sec. - res.: 1024** $\times$ **768**
Gaming	A typical game-sequence in **Unreal Tournament 2004** where robots play with each other, from the spectator viewpoint. OpenGL GPU-acceleration is used (GeForce 6800 video-card). **time: 240sec. - res.: 1024** $\times$ **768**
Video	A H264/AVC pre-recorded fragment of a football match at 25 fps, played with **mplayer**. (without GPU-acceleration, to be usable on classic thin-clients). **time: 140sec. - res.: 640** $\times$ **480**
Browsing	A sequence of 20 webpages is shown automatically (with Javascript) in **Mozilla-Firefox** with a gap of 5 seconds. The webpages are saved on a local webserver and include fixed images but also flash animations.**time: 100sec. - res.: 1024** $\times$ **768**

delay is 10% higher for the video scenario and 20% lower for the office and browsing-scenarios (due to higher resp. lower complexity).

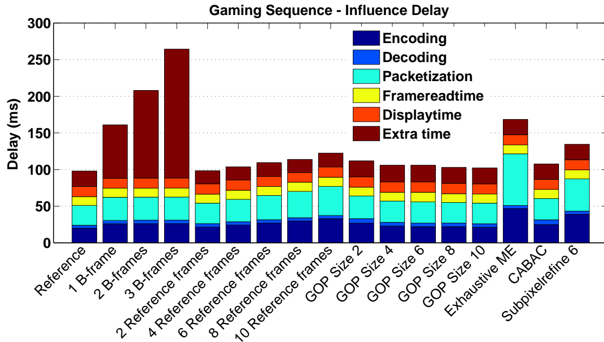

Figure 6: Influence on delay for the most determining parameters.

Nevertheless, the delay is still too high (the minimum measured delay is 100 ms) but advances in hardware will be able to reduce encoding and decoding time significantly. Also, the first lines of the next frame are already buffered in the framegrabber during the encoding of the current frame. As a result the interaction delay as experienced by the users will be around 40ms higher for each scenario than shown by figure 6. By reading out frames in blocks and encoding them piece by piece, the extra delay-overhead can be avoided. This will give similar encoding results, because a motion estimation search range of only 4 pixels gives in all scenarios optimal results. The optimal encoding can be performed only by taking neighbourhood macroblocks into account.
We first want to determine the minimum GOP-size for each scenario. With lower GOP-sizes, video quality degredation will be less in the case of packet loss for streaming video. As expected, the number of I-frames has most influence on the bandwidth per frame for the low motion scenarios (browsing and office) due to large similarities between succeeding frames (see figure 7 (a) and (b)). For the low motion scenario, a GOP-size of 60 frames has almost no influence on delay and bandwidth per frame; for high-motion scenarios however, a GOP-size of 40 frames already gives optimal results (the rate distortion effects are overwhelmed due to frequent scene changes).
For the streaming desktop, B-frames can only be used when the server detects no user interaction for a certain time (e.g. when the user is watching a video) because delay is augmented by the number of B-frames multiplied by the (framereadtime+1) due to buffering. Only for very low-motion scenarios (office in our case), it is usefull to use 2 B-frames (due to large similarities between succeeding frames). For all other scenarios, 1 B-frame suffices. As soon as the user interacts with his thin-client, the usage of B-frames should be turned off immediatly.
The number of reference frames used for P and B-frame encoding is important when there are a lot of *repeating elements*. When more than 4 reference frames are used, delay increases by 10% or more and the bandwidth-gain is minimal (see figure 6). Only in the video-scenario, there seem to be enough repeating elements to profit from using 2 extra reference frames. For low-motion scenarios, the optimal residu can be found only by looking at the previous frame, and it is not necessary to keep extra reference frames (see figures 8 (a) and (b)).

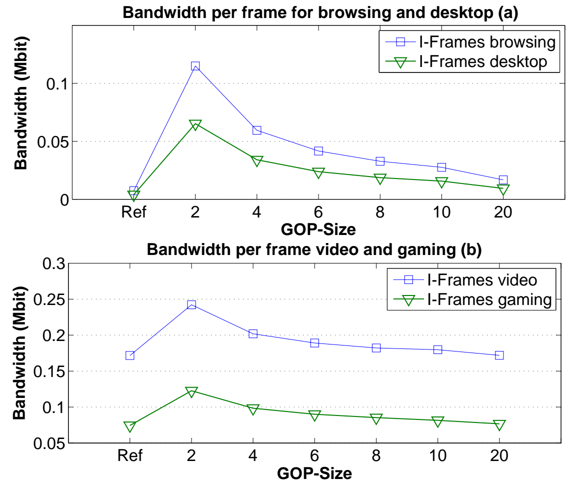

Figure 7: Influence on the bandwidth per frame for different GOP-sizes.

In addition to the results shown in figures 7 and 8 we also measured a number of other parameters. The use of CABAC can reduce the amount of bandwidth per frame up to 20% for high-motion sequences and up to 10% for low-motion sequences. However, there is a delay overhead of 10% and CABAC should only be used when there is enough processing power. An exhaustive motion estimation search algorithm only showed a bandwidth gain (up to 8%) in the gaming scenario (due to the high amount of detail). Ex-

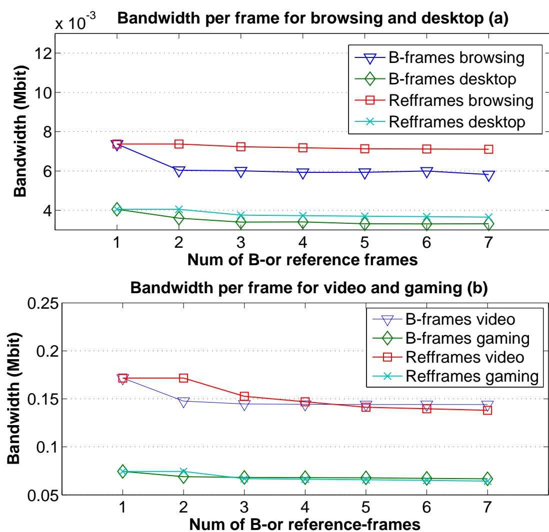

Figure 8: Influence on the bandwidth per frame for a different number of B-frames and reference-frames.

haustive search is currently too slow to be used. Using smaller macroblock partitions for motion estimation can reduce bandwidth with 10% for high motion sequences due to a large amount of detail, but the delay overhead is too large to be usefull.
Furthermore, we have also tested the influence of other motion estimation algorithms, subpixel motion estimation and the motion estimation range but they did not show a significant reduction in bandwidth. Adaptive and weigthed bi-predicted B-frames showed only minor differences with fixed B-frames.
In our final tests, we combine the optimal parameters. For the low-motion scenarios, we use a GOP-size of 40, turn CABAC on, the full range of macroblock partitions (up to 4 $\times$ 4 pixels) and one extra reference frame. We test the high-motion scenarios with the same settings except the GOP-size (60 instead of 40) and the number of extra reference frames (2 instead of 1). We also test all four scenarios with these settings and 1 extra B-frame. Figure 9 summarizes the results. As our streamer PC was a little too slow to encode 25 frames per second, we multiplied the bandwidth per frame with 25 to get an idea of the theoretical possibilities of realtime desktopstreaming. The video-scenario shows a reduction in average bandwidth of 30% if 25 fps are sent using the optimal encoding settings without B-frames. For the other scenarios, the bandwidth-gain is smaller, also indicating the video encoder parameters are strongly correlated. Figure 9 (b) shows bandwidth requirements are only moderate for each scenario, even with the reference settings. Only 2 Mbit/s is needed to stream a recent 3D Game at a resolution of 640 $\times$ 480 to a thin-client. Video requires most bandwidth (4 Mbit/s). As delay is much less important for video, bandwidth can be reduced to 3 Mbit/s by using the optimal settings. The office and browsing-scenarios only require 115 Kbit/s resp. 185 Kbit/s when the reference settings are used. We expect that with future hardware encoders, it will be possible to stream in realtime any possible desktop scenario with only moderate bandwidth requirements. However, servers should be placed close enough to the end-user to keep interaction delay within reasonable limits.

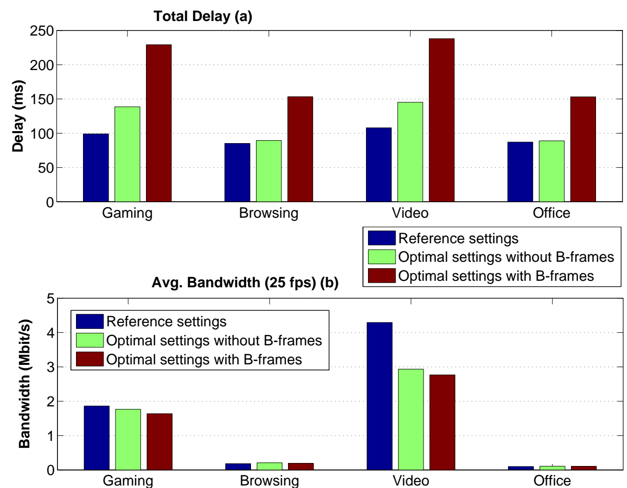

Figure 9: Delay and bandwidth of desktopstreaming with optimal encoder settings

4.3 Comparison with classic thin-client systems

To compare classic thin-client systems with realtime desktopstreaming, we measured the total amount of data transferred for both low-motion scenarios and the high-motion video scenario. For FreeNX we used the ADSL compression setting and for tightVNC all features were enabled. As only an older version of THiNC was available, we only compared the browsing and video-scenario. For the bandwidth- comparison of the video-scenario, the fragment was played back with 1 frame per second (To make sure all data is transferred as e.g. VNC has a client-pull architecture). The total amount of traffic for the office and browsing-scenario is low enough to be sure no data is discarded.
With desktopstreaming, the low-motion scenarios originally run at a resolution of 1024 $\times$ 768 were scaled down to 640 $\times$ 480 by the framegrabber, while the classic thin-client systems show a full-screen resolution of 1024 $\times$ 768. To compare bandwidth, we multiplied the total amount with 2.56. A small experiment showed this is indeed a good approach. Figure 10 shows the total amount of data transferred for the video scenario (in Mbit). It is clear desktopstreaming outperforms the other approaches.
Both low-motion scenarios (also see 10) however use equal or less bandwidth than desktopstreaming if FreeNX is used. If we also look at the required processing power, FreeNX is a better solution than desktopstreaming. Desktopstreaming uses for low-motion sequences around 50% CPU on encoding side and 30% CPU on decoding-side, while classic thin client systems only use a maximum of 5% CPU on both encoding and decoding-side. If we take processing-overhead for the video scenario into account, classic thin-client systems are in the same order of magnitude as desktop streaming. The arrows in figure 10 show that with a hybrid thin-client protocol using a combination of FreeNX and videostreaming, optimal results will be obtained in all scenarios.

To compare the image quality, we also played the video fragment with 25 fps. We used the formula (4.1) proposed

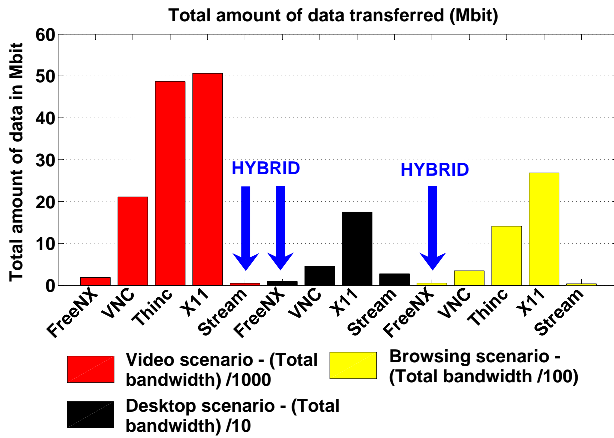

Figure 10: Comparison of the total amount of data transferred for 3 scenarios for different architectures.

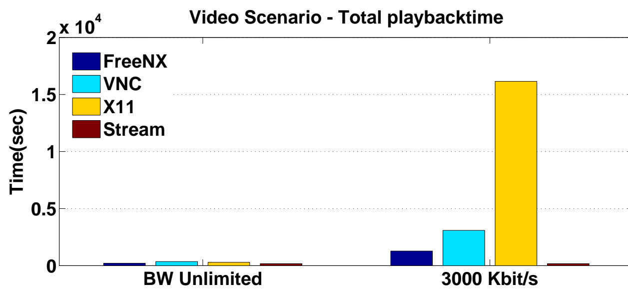

Figure 11: Playbacktime of the video scenario when played with 25 FPS for different architectures.

and explained in [5] to calculate the video quality percentage. A first test was performed without bandwidth limitation between client and server, a second test with a limitation to 3 Mbit/s as a constant bitrate of 3 Mbit/s for videostreaming is enough to get good video quality (also formally confirmed by the PSNR-ratio, not included here). To calculate the video quality percentage, we need the playbacktime (see figure 11) and the amount of data transferred when the fragment is played with 25 FPS (see figure 12). For videostreaming, the amount of data transferred when played with 1 fps is the same as when played with 25 fps. The only penalty there is the playback-time (as we are playing the videofragment with only 20 fps). Figure 13 shows the results. A videoquality of 100% is the optimal quality, the same as when a video is played back locally. We clearly see the streamingdesktop has still a very high quality (even with re-encoding) of 80% if bandwidth is unlimited and 70% with limited bandwidth, proving desktop streaming is ideally suited for multimedia applications.

$$VQ(P) = \frac{\frac{Data(P)/PlaybackTime(P)}{IdealFPS(P)}}{\frac{Data(slowmo)/PlaybackTime(slowmo)}{IdealFPS(slowmo)}} \quad (4.1)$$

5. CONCLUSION AND FUTURE WORK

We developed a realtime desktopstreamer which uses a

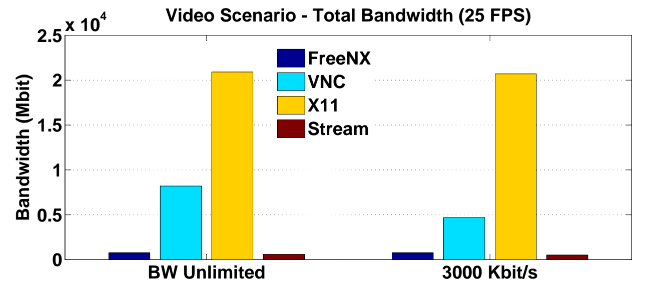

Figure 12: Total bandwidth for the video scenario when played with 25 FPS for different architectures.

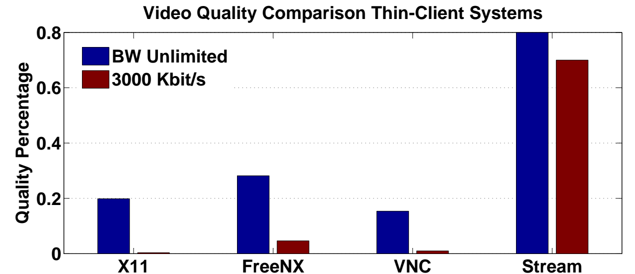

Figure 13: Image quality comparison for the video scenario using the video quality percentage formula (4.1).

H264/AVC-based videocodec to stream the graphical output of applications to a thin-client device. We compared this approach to classic thin-client systems which are best suited, especially if processing overhead is taken into account, for office programs and webbrowsing. When multimedia or 3D games are used, the realtime desktopstreamer outperforms classic thin-client systems and is the only viable solution. Videostreaming can offer a solution for thin-client systems, especially if fast low-power video-encoding and-decoding chips will become available. We propose to use a hybrid protocol using a combination of a classic thin-client protocol and desktop streaming. In our future work, we will perform an evaluation of the energy consumption and do a processing power analysis of classic thin-client systems and desktop streaming, to validate their usability on portable devices.

6. REFERENCES

[1] L. Aimar, L. Merritt, et al. *x264 - A free H264/avc encoder*. http://developers.videolan.org/x264.html.

[2] R. Baratto et al. Thinc: A virtual display architecture for thin-client computing. In *Twentieth ACM symposium on operating system principles*, 2005.

[3] J. M. Danskin et al. Fast higher bandwidth x. *Multimedia and Network*, pages 192–199, 1995.

[4] J. Liesenborgs. *JRTPLib, an object-oriented RTP-library written in C++*, 2006. http://research.edm.luc.ac.be/jori/jrtplib/jrtplib.html.

[5] J. Nieh, S. J. Yang, N. Novik, et al. Measuring thin-client performance using slow-motion benchmarking. *ACM Transactions on computer systems*, 21:87–115, 2003.