Automatic fine-grained area detection for thin client systems

Bert Vankeirsbilck[a], Dieter Verslype[a], Nicolas Staelens[a], Pieter Simoens[a,b], Chris Develder[a], Bart Dhoedt[a], Filip De Turck[a], Piet Demeester[a]

[a]*Ghent University - Department of Information Technology (INTEC), IBBT*
Gaston Crommenlaan 8, bus 201, 9050 Gent, Belgium
[b]*Ghent University College - Department INWE, Valentyn Vaerwyckweg 1, 9000 Gent, Belgium*

Abstract

The widespread availability of cloud infrastructures is fueling new interest in the thin client computing paradigm. However, current thin client protocols are not designed to handle new content types as often encountered in state-of-the-art applications (e.g. multimedia editing, gaming, multimedia playback). Conveying this content using traditional thin client protocols typically results in a combination of excessive resource usage and low visual quality. In this paper, we propose an approach where the content type can vary for different portions of the screen (e.g. combination of static text and video). Once the different content types have been detected, each of them can be encoded using the most appropriate algorithm. We present two algorithms for this runtime detection. The first algorithm is operating at the pixel level, thereby being independent of the actual thin client protocol used. The second algorithm assumes the presence of a rectangle-based thin client protocol (such as the popular VNC protocol), trading independence for improved performance. The appropriate parameter settings for these algorithms are experimentally determined. Furthermore, their influence is studied in detail in terms of detection accuracy, and the time to perform the algorithms is analysed. Accurate hints are derived within less than 10 ms, indicating the high potential of this approach for use in next generation thin client systems.

Keywords: Thin Client, Area Detection, Encoding, Differentiation, Content classification

1. Introduction

The thin client computing paradigm consists of offloading as many resource intensive operations as possible to a remote server and limits the client functionality to forwarding user events to the server and displaying the returned graphical content. The server handles complex computations and remote storage. The benefits from this approach are numerous and well-known [1], including decreased exploitation cost, prevention of data loss, shared hardware resources, easier data exchange, universal access of applications, easier software update management and maintenance, etc. Clearly, the currently witnessed increase in cloud deployment is a strong enabler to leverage the functionalities and possibilities of the constrained devices such as tablets and smartphones, used to access these cloud infrastructures. Hence, a cloud architecture based on thin client computing can provide access to data and applications from any place, at any time with any device and most importantly with unhampered user experience.

Given the current network connectivity and thin client protocols to convey information between client and server, the delivery of dynamic visual content such as games and video is often an issue. Thin client protocols are typically optimized for delivering static content, and adopting the same approach for delivering dynamic content results in excessive bandwidth usage and mediocre visual experience. An approach frequently adopted in commercial thin client protocols consists of redirecting multimedia content directly to the client device for local decoding. The concept is used in Microsoft Remote Desktop Protocol (RDP) version 7 [2] and the High Definition User Experience (HDX) MediaStream feature in Citrix' products [3]. Although this approach works well for supported video formats, it is not applicable to applications that do not use a video playback codec for their visualization (such as games and visualization software).

Email address: Bert.Vankeirsbilck@intec.UGent.be (Bert Vankeirsbilck)

To alleviate these problems, we have previously proposed a hybrid approach [4] that switches between two modes of operation: the classic thin client mode (suitable for static content) and a media streaming mode (preferable for dynamic content). However, a single encoding is used for the complete screen, leaving substantial room for further improvement by applying different encoding methods for different areas of the screen.

The contributions of this article can be summarized as follows. We first identify the gains (in terms of bandwidth efficiency and visual quality) of differentiating dynamic from static content on a fine-grained scale, i.e. by applying an optimized encoding for each display area. Secondly, two algorithms to perform the detection of different content types are proposed and evaluated in terms of accuracy and performance. One algorithm purely operates at the pixel level which is the lowest layer in the rendering stack, thus being highly system independent. The second algorithm uses higher level information from a thin client protocol (more specifically information on rectangles to be displayed), assuming the availability of this information, thereby trading system independence for performance (specifically in terms of analysis speed and runtime overhead). Finally, these algorithms, together with their appropriate parameter settings, are thoroughly studied to highlight the potential of these novel approaches for next generation thin client systems.

The remainder of the manuscript is organized as follows. Section 2 presents an overview of related research efforts addressing fine-grained area detection for thin client systems. In Section 3, we propose a generic architecture and building blocks to enable the classification of screen content. Two strategies to differentiate dynamic from static areas are discussed in Section 4. Section 5 presents results obtained through experiments with proof-of-concept implementations. Conclusions are summarized in Section 6.

2. Related work

In a thin client system, remotely executing applications heavily use the server side graphics Application Programming Interface (API) to render the content they generate. Subsequently, this graphical information is intercepted at a particular level in the visualization stack, compressed, encapsulated and transmitted over the network. The problem with this approach is that the application output is already processed for visualization and the server side of the thin client protocol is unaware of the exact nature of the generated graphics. This leads to the fundamental problem that the encoding of the content is performed agnostic of the type of graphics that is displayed. The most visible result of this approach is the encoding of embedded video in a pixel-format, leading to excessive bandwidth usage and mediocre playback quality. The performance and issues with classic thin client computing protocols have been investigated thoroughly in [5, 6, 7], indicating the need for designing intelligent thin client protocols given that the user experience depends on network latency and bandwidth.

The problem with multimedia has been partly addressed in newer thin client protocol versions. Virtual Network Computing (VNC) [8] and FreeNX [9] include audio support. Microsoft RDP 7 and Citrix provide a similar solution (i.e. Windows Media Player redirection and HDX MediaStream respectively) by intercepting video files at the OS level and forwarding (possibly after transcoding) this content directly to the client for playback. However, for other dynamic content that is not rendered through a video codec (such as games), the screen rendering must still happen at server side and will cause a considerable burden on the network [10]. Red Hat's SPICE protocol [11] mentions heuristic identification of video streams to transmit them as M-JPEG video streams, while Teradici PC over IP (PCoIP) does feature detection of text, graphics and video on screen and encodes these accordingly to save bandwidth, as described in [12]. As these solutions are proprietary, no further information on these algorithms is available.

Previously [4] we have proposed a remote rendering architecture that differentiates screen content such that the whole screen area is either transmitted as a video stream or through a classic thin client protocol encoding. In the current article we extend this concept, aiming at finer-grained detection of dynamic and static regions, to enable encoding of parts of the screen as classic thin client protocol encoding and others as a video stream simultaneously.

Recent research [13] hooks into the Linux X server and monitors X protocol messages, resulting in an OS dependent strategy to differentiate video content from static content. By correlating image drawing commands (e.g. putImage) per application and monitoring their

frequency, complete applications are classified as visually dynamic or static. The knowledge of these application windows' size and position on screen is used to differentiate between regions in the screen. The authors show that this approach is computationally beneficial, but also take the assumption that the presence of all dynamic content can be detected based on repeated invocations of the putImage()-method. However, not all applications rely on putImage() calls to render their content (e.g. GPU assisted applications). In addition, some applications also use putImage() to render static content, more specifically to refresh large portions of the display at high frequency to ensure fast updates (e.g. in games). Our proposed system focuses on pixel level analysis, where the actual screen content can be more accurately classified. By performing this analysis at the pixel level, independence on other platform components (e.g. graphics API, underlying OS and type of thin client protocol) is achieved.

Currently available standards concerning representation of scenes and meta-data formats have built-in support for content type differentiation. The MPEG standards support this differentiated content by offering multiple encoding channels in their scene formats, e.g. Binary Format for Scenes (BiFS) and Lightweight Adaptive Scene Representation (LASeR), both part of the MPEG-4 standard [14]. Thus, a thin client system using MPEG streaming could be developed based on the work presented in this paper.

The concept of thin client based cloud computing is an active research area [15], and has been found interesting for enabling games on resource constrained devices [16, 17]. In the current paper we focus on an important building block of such systems and fill a crucial algorithmic gap to classify screen content for optimizing user experience and bandwidth consumption.

3. Hybrid encoding architecture overview

3.1. Content analysis component

Figure 1 presents the hybrid thin client encoding architecture, with an analysis component steering the encoding process. The analysis component retrieves information from various sources in the thin client server, as will be elaborated on further in this section. Based on this information, hints are given to the thin client encoder about the content type of individual

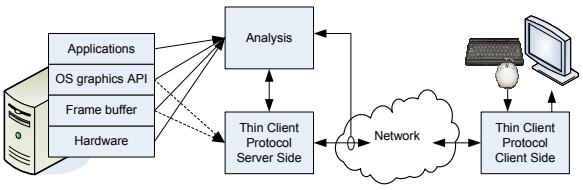


Fig. 1: Hybrid encoding architecture.

components, such as parts of the screen that are text, dynamic content, available in the client side cache [18], vector graphics encoded or to be fetched at the client side from another source or from the client hard drive. The thin client protocol encoder subsequently fetches the required input, encodes this content in the appropriate format according to the hints provided. Thus, the thin client server continuously sends appropriately encoded screen content to the client, while the analysis component is responsible for configuring the size and position of the differently encoded regions. The analysis component operates independently from the thin client encoder, for multiple reasons. First of all, this approach allows for hysteresis in the analysis, i.e. that the output of the algorithms can be held in deliberation for a period of time until it stabilizes, before reconfiguring the encoding parameters. The decoupling also ensures that the analysis algorithms can be used irrespective of the specific thin client protocol actually used. Furthermore, multiple encoding threads can be active at the same time, each taking care of the encoding of one or more specific regions and operating with its own dedicated parameters. For instance, an advanced analysis scheme could detect 3 regions of high motion content with different encoding needs. Three separate encoding threads could then be activated, with settings adapted to the specific content of the regions, e.g. one encoder operating at 60 fps allowing slight quality degradation, and the other at 30 fps with perfect quality. With the independent operation of analysis and encoding, the analysis component could be set to operate at 5 fps, implying that not each and every frame is subject to detailed analysis. This can limit the computational overhead of the overall system, keeping resources available for the graphics encoding.

3.2. Overview of interception options

As depicted in Fig. 1, the content detection could retrieve information from all layers of the visualization stack.

- At application level, the programmer could explicitly indicate when and where dynamic content is generated. The information on this level is expected to be very accurate. However, relying on this application level information would limit the support for legacy applications since explicit input from the application developer is needed. Furthermore, (partially) hidden application windows would need to be monitored, as an application is generally unaware of window manager functionality.

- The OS level provides a graphics API that is used by applications to draw to the screen. Information can be fetched about drawing methods to track frequently updated graphics. As mentioned before, this route has been explored in case of the Linux operating system [13], and although being effective, it also has considerable drawbacks (more specifically dependency on specific API calls and operating system are introduced). More importantly, since the applications are not developed to execute over a (possibly high latency) network, they are often designed using a greedy updating strategy (i.e. updates are generated at the highest possible frequency, possibly leading to superfluous updates). This is typically the case for visualization software and games that tend to redraw the content as fast as possible while it is being generated.

- At frame buffer level, the graphic information that should be represented at client side is available as pixel data, and is in fact the only true source for analysis of the dynamics of the on-screen behavior. In the thin client server, a common optimization is to track the modified region so that the transferred graphical update represents only what has changed on the screen in comparison with the previous update. The information can be considered very low level, since it concerns raw pixel information. While other interception points have more condensed and meta-information-rich formats, analysis on the frame buffer level has the drawback of being computationally intensive. Furthermore, there is possibly a tendency of overestimating motion regions, given that small areas can accidently be classified as motion content. The latter also tends to lead to fragmentation of detected motion regions, causing the analysis complexity to increase.

- At hardware level, one could monitor the area on screen to which graphics support (e.g. GPU or video decoding) is granted and decide that these areas are high motion, or hook into the hardware rendering driver as in [19]. However, besides having static applications with GPU support, the diversity of existing hardware makes universal application of such an algorithm unfeasible in practice.

- Another source of information that can be taken into account for deciding on the encoding mode is that of network statistics. As reported in previous work [4], miscategorization of screen contents leads to an increased usage of network bandwidth and computing resources. At network level, a straightforward strategy would be to switch between streaming and classic thin client mode based on bandwidth measurements, but this only provides aggregate information on the complete screen instead of identifying particular regions.

In conclusion, depending on the source of the information the analysis component bases its decision on, the hinting accuracy will differ, as will the genericity of the analysis algorithm.

4. Area detection strategies for visual content

In this section we present and analyse the two most promising detection approaches from the interception possibilities highlighted above: the first algorithm operates at the pixel level, while the second approach assumes information of rectangle updates made available by a separate system. Algorithms for both strategies are discussed, and their algorithmic complexity is deduced to indicate the parameters that influence their performance.

4.1. Pixel analysis

Figure 2 depicts the proposed algorithm for realizing a fine grained classification of screen content via pixel analysis. The screen content is sampled in order to speed up the analysis process. For this sampling, one pixel is taken to represent multiple surrounding pixels. This way, the number of pixels to analyse is lowered, simplifying all following steps that depend on that number. However, since one pixel represents a set of different pixels, the accuracy is obviously expected to be affected. The number of changes is recorded for each sample, and a threshold is chosen to distinguish static from dynamic sample areas. A blob detection algorithm is subsequently used to construct rectangular

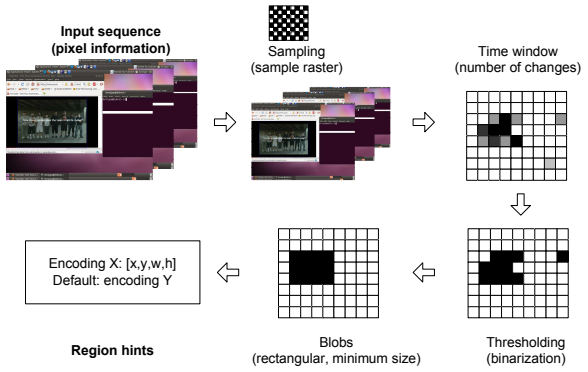


Fig. 2: Schematic representation of the pixel analysis algorithm.

areas that are marked "static" or "dynamic", which are used as hints to the thin client protocol. Blob extraction basically searches regions in images, by computing the similarity of all pixels to a selection of their direct neighbours, and deciding whether they are part of the same visual object or rather form an independent part in the image. This way, the image is divided into groups of pixels, called blobs, which are considered to form a unity.

Listing 1 Pixel analysis algorithm

```
1:  DiffFrame_N = Sample(Image_N − Image_{N−1})
2:  for all pixels i ∈ DiffFrame_N do
3:      if DiffFrame_N(i) ≠ 0 then
4:          windowMatrix(i) ++
5:      end if
6:  end for
7:  for all pixels i ∈ DiffFrame_{N−w} do
8:      if DiffFrame_{N−w}(i) ≠ 0 then
9:          windowMatrix(i) −−
10:     end if
11: end for
12: for all counters ∈ windowMatrix do
13:     if counter ≥ motionThreshold then
14:         binarized(counter) = 1
15:     else
16:         binarized(counter) = 0
17:     end if
18: end for
19: return blobAlgorithm.computeBlobs(binarized)
```

Listing 1 presents this pixel analysis algorithm in pseudocode format. The first line represents the creation of a difference frame between the screen content at time N and the previous screen update (at time $N-1$). The resulting difference is sampled according to a sampling raster and constitutes a difference frame for the screen update at time N. Lines 2 to 11 deal with the maintenance of the time window of a given window size w, represented by a *windowMatrix*. All pixels of the current difference frame are evaluated. For pixels that have changed since the last screen update, a corresponding counter in the *windowMatrix* is incremented. Since the *windowMatrix* concerns updates from a given windowsize w, the counters corresponding to the content of the difference frame representing the changed pixels at time $N-w$ need to be decremented. Lines 12 to 18 describe the thresholding. All *counters* in the *windowMatrix* are checked against the *motionThreshold*. Values under this threshold are classified as static pixels, values above are classified as motion pixels, represented in a *binarized* matrix as 0 and 1 respectively. Finally, in line 19, *blob extraction* is performed with this *binarized* matrix resulting in unambiguous hints about the division in motion and static content.

Clearly, sampling the image and updating the counters is of complexity $O(res \times S)$, where *res* represents the resolution expressed total number of pixels while S stands for the sampling rate *(*with $0 < S \leq 1$). In addition, N sampled difference frames should be kept in memory, allocation of which requires $O(res \times S \times N)$ operations. Therefore, lines 1 to 18 require $O(res \times S \times N)$ operations. The complexity of the blob detection step depends on the particular blobbing algorithm that is used. Connected Component Labeling (CCL) is an algorithm that is frequently used in digital image processing. The algorithm uses two variants of patterns for modeling connectivity between neighbouring pixels, i.e. 4 connectivity where only the pixel to the North and West are considered to determine a pixel's label and 8 connectivity where the pixels North-West and North-East are considered as well. As described in [20], the original algorithm featured a two-pass approach but has been optimized to a single-pass version for streaming video processing. If the latter is used, the blob algorithm evaluates all pixels of the binarized window matrix once, leading to $O(res \times S)$ complexity. As proven in [21, 22], the CCL algorithm scales linearly with the number of pixels to be analysed. The resulting overall complexity of the algorithm is $O(res \times S \times N)$.

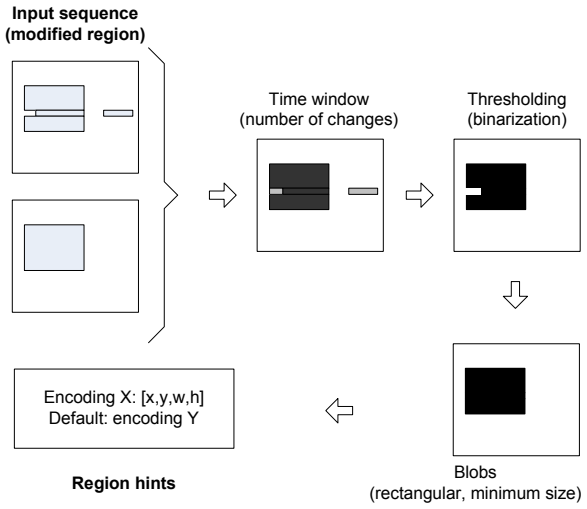


Fig. 3: Schematic representation of the thin client protocol rectangle analysis algorithm. In contrast to Fig. 2, the input sequence is in the form of modified regions, resulting in an algorithm that operates on different data.

4.2. Thin client protocol rectangle analysis

We assume a thin client protocol to be rectangle-based and to have knowledge about what has changed between consequent updates, i.e. modified regions are specified as a set of rectangles (this approach is adopted by VNC [8]). Therefore, hints are received from the OS about the modified areas on screen. This is an approach that most thin client protocols use in order not to encode the full screen, thus saving bandwidth and unnecessary computing cycles. The thin client protocol rectangle analysis algorithm depicted in Fig. 3 augments this approach and uses it to find frequently updated areas on the screen. Over a time window of given size, the number of changes of the pixels are recorded. Using a threshold to classify motion and static pixels, binarization of the time window is achieved, which can be fed into the blob detection algorithm to extract rectangular areas to construct the hints for the thin client protocol encoder.

Lines 1 to 6 in Listing 2 relate to maintaining the time window, which is represented by a *windowMatrix*. For this rectangle analysis, the rectangles in the update at time N represent the changes with respect to the visual content at time $N-1$. In the *windowMatrix*, the corresponding counters for the rectangles are incremented. Also, the counters corresponding to rectangles from the update that are out of scope of the time window i.e. at time $N-w$, need to be decremented. Lines 7 to 13

Listing 2 Thin client protocol rectangle analysis algorithm

```
1:  for all rectangles ∈ update_N do
2:      windowMatrix.add(rectangle)
3:  end for
4:  for all rectangles ∈ update_{N−w} do
5:      windowMatrix.remove(rectangle)
6:  end for
7:  for all rectangle ∈ windowMatrix do
8:      if rectangleupdatecount ≥ motionThreshold
        then
9:          binarized(rectangle) = 1
10:     else
11:         binarized(rectangle) = 0
12:     end if
13: end for
14: return blobAlgorithm.computeBlobs(binarized)
```

concern the thresholding step of the analysis. As with the pixel analysis, a *binarized* matrix is constructed representing values above the *motionThreshold* by ones and values under the *motionThreshold* by zeroes. Finally, the results are computed from this *binarized* matrix in line 14 through *blob dectection*.

To assess the complexity of this algorithm, we note that the complexity of maintaining the data on the rectangular updates (lines 1 to 6) amounts to $O(R)$, with R representing the average number of rectangular updates for each screen update. Determining the static or dynamic behavior of each sampled pixel (lines 7 to 13) requires $O(res \times S)$ operations, while maintaing this information over N timeframes requires $O(res \times S \times N)$. As mentioned before, computing the blobs happens in linear time, so the overall complexity of the algorithm is $O(res \times S \times N)$. However, we expect the average case behavior to be considerably better than for the first algorithm, since advantage can be taken of the rectangular updates (allowing for more efficient classification than in the pixel based analysis).

4.3. Qualitative trade-off discussion

Comparing the pixel analysis with the thin client protocol rectangle analysis, the pixel analysis has higher potential to perform accurate detection of motion in a graphical session. Since it uses information about the actual changes between screen updates, it is able to monitor for each pixel exactly how often it changes. For the thin client protocol rectangles, the information comes from the operating system, that gives hints on

what area of the screen that has been redrawn. However, this does not necessarily mean that the content of this modified area has changed. For instance, if a video containing solid black frames is displayed, the video player will request the operating system to draw each frame, since the video player is unaware of the nature of the video it is playing. The OS will do so, and thus will also mark the area of the video as modified although visually for the user, nothing has changed. The same holds for padding black rectangles of videos to scale to different resolutions. This observation means that the thin client protocol rectangle analysis will perform an overestimation of modified areas between updates. The impact of this behavior will be high for applications that focus on achieving high frame rates, such as games and all kinds visualization software. These applications are most often implemented using a visualization loop that aims to redraw the content of the screen as fast as possible, irrespective of the content that needs to be drawn. Pixel analysis overcomes this issue, hence the potential for higher accuracy over thin client protocol rectangle analysis.

5. Experimental results

Both approaches (i.e. pixel analysis and rectangle analysis) were prototyped in C, and evaluated in terms of their detection accuracy. Besides presenting the experimental environment and the applied measurement methodology, the following most prominent research questions in this section are addressed:

- Does fine-grained detection offer significant advantages in terms of bandwidth requirements and visual quality (5.2)?
- Which are the effects of the parameters of the proposed detection algorithms (5.3)?
- Is real-time analysis with the proposed algorithms possible (5.4)?
- How do the algorithms perform under new scenarios and for larger resolutions (5.5)?
- How do inaccuracies of the detection algorithms manifest themselves (5.6)?

5.1. Setup

An H.264 encoding type was added to Tight VNC version 1.3.10, using the x264 encoding library through ffmpeg at server side, and ffmpeg for decoding at client side. The x264 ultrafast, zero latency profile was used. Dynamic control over this adapted VNC system was enabled, such that multiple dynamic regions can be defined via a simple TCP-based message protocol. Finally, the output of the designed analysis component is coupled to this interface. Table 1 summarizes the hardware and software used for the experiments, Fig. 4 presents the setup that was used.

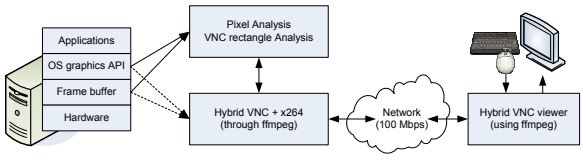


Fig. 4: Experiment setup.

Table 1: Hardware and software configuration used for experiments.

Server	Pentium®Dual-Core CPU E5400 @ 2.70 GHz 4 GB RAM
Client	Intel®Core™2 Duo CPU P8600 @ 2.40GHz 4 GB RAM
Video codec H.264 encoding	ffmpeg version SVN-r20467 x264 library version 0.78, ultrafast, zero latency profile
Thin client system	Tight VNC version 1.3.10
Blob detection library	IPL98 version 2.20

The network traffic was monitored using *tcpdump*, avoiding interference of the network monitoring with the data transfer. The frame rate was measured at the VNC viewer, which has been instrumented (the method where content is written to the screen is adapted). This influence is actually negligible since we can acquire a maximum frame rate of 55 frames per second with a viewer that requests frame updates as fast as possible, for a resolution of 1920×1080. At times the original frame sequences were to be recorded, the server writes the screen content to a file since we have taken the analysis offline in our experiments. The machine we performed this capturing on, is equipped with a solid state hard drive, of which we have measured a writing throughput of 248 MB/s using the Linux command line tool *dd*. The graphical updates are written to that solid hard hard drive in raw format, i.e. 3 Bytes per pixel. For a resolution of 1920×1080 and at a rate of 30 frames per second, the needed throughput towards the

hard disk is (1920 × 1080) pixels/frame × 3 B/pixel × 30 frames/s = 186.624 MB/s which is well under the available 248 MB/s. Also, during our capture, we monitored the CPU load of the server and assured it stayed well under 100%, and could also monitor the frame rate of writing to the screen by the viewer and assured that 30 fps were maintained. Screen captures at the client side were performed from within the viewer, by means of executing Linux screen shot utility *scrot* after each screen update operation, thereby specifically capturing the output of the vncviewer application. Since these captures were only performed for low motion scenarios, the impact on interactivity and frame rate was negligible.

5.2. The penalty of misclassification

Misclassifying visual content causes suboptimal encoding, leading to either excessive bandwidth usage (due to inappropriate compression) or a severe visual quality reduction, or possibly both. When an appropriate encoding scheme is adopted for a given visual content item, ideally the bandwidth usage would be low and the resulting visual quality high. In practice however, often a trade-off is to be made, sacrificing some bandwidth to improve visual quality. This trade-off is investigated for the hybrid streaming/VNC encoding approach.

To investigate the effect of classifying content either as motion (and stream it with a video codec) or static (and transfer it using a thin client protocol), we assessed both the bandwidth, achievable frame rate and visual quality using manually configured encodings for known content. We therefore recorded a desktop session combining static areas and a single video, for which the size was varied between 0% and 100% of the screen of 1024×576 pixels. (The screen resolution was chosen to equal that of the video source file, as to avoid the need for upscaling or padding.) The manually configured encoding was either (i) full screen video streaming, representing classification of the entire screen as motion, (ii) full screen VNC encoding, representing non-motion classification, and (iii) a perfect detection. For the latter, since the location for video playback is configured we know the exact size and position of the video (and hence no classification algorithm is used), but directly configure the encoder to stream exactly the video rectangle, and use VNC for the remaining static content. Thus, the latter serves as a theoretical approach assuming perfect classification, for benchmarking purposes.

Using the "Big Buck Bunny" (http://www.bigbuckbunny.org) movie with a playback rate of 24 fps, the consumed bandwidth was measured, together with the frame rate achieved and the resulting screen update quality for each motion region size. Each experiment was executed 10 times to reduce measurement noise.

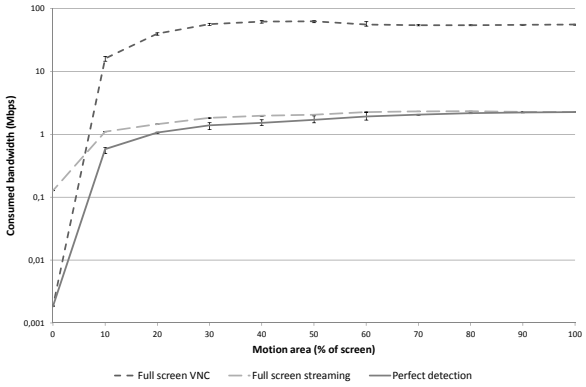


Fig. 5: The effect of misclassification on bandwidth: bandwidth consumed for transmitting a scene as a function of motion region size, with different unoptimized encoding strategies in comparison to perfect detection. Measurements for 1024 × 576 screen resolution. The error bars indicate the minimum and maximum of the recorded values.

Figure 5 presents the average bandwidth used to transfer the graphics of a session for varying area size of motion content. The figure clearly shows that VNC consumes considerable resources and is therefore less suited to support this scenario. However, the protocol is very well suited for static content and consumes no bandwidth when no screen updates take place. The approach used by VNC for encoding screen content is to divide a screen update in rectangles that are individually compressed and transmitted to the viewer for presentation. In the first part of the graph the consumed bandwidth raises proportionally with the video size since more pixels need to be encoded. At a given point this behavior changes since the rectangulation and encoding consumes more time, causing a drop in rate of encoded screen updates, as visible in Fig. 6. While providing this degraded visual performance, the consumed bandwidth of VNC encoding exceeds 60 Mbps. Depending on the overestimation of the motion area, full screen streaming consumes more bandwidth compared to perfect detection. In addition, the dedicated encoding approach ensures optimal delivery quality for both content types, as visible in Table 2. Since for larger video sizes, motion vectors can be used efficiently for video coding, both graphs saturate. The maximum consumed bandwidth for video streaming is 2.33 Mbps, and is reached at 70% motion area size.

The overhead in comparison with 100% motion area size is mainly caused by the static region confusing the motion vectors onto which the video codec is based to compress efficiently. For this experiment, the upper bound on standard deviation for all measured values of consumed bandwidth amounts to 3.78 for full screen VNC encoding, 0.02 for full screen streaming and 0.21 for perfect detection.

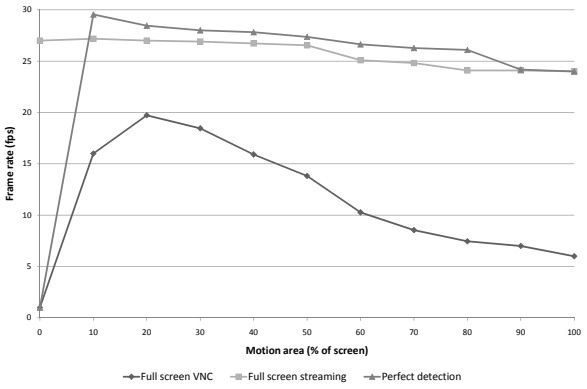


Fig. 6: The effect of misclassification on frame rate which has a direct impact on user experience. Measurements for 1024×576 screen resolution.

Figure 6 shows the frame rate that is reached as a function of the size of the motion region on screen. With 0% motion, VNC only delivers graphical updates when user interaction occurs. In our experiment, this resulted in a frame rate of 1 fps, while the full screen streaming encoding delivers this static content at an unnecessary 26 fps. When motion content is visible on screen, the frame rate VNC encoding manages to offer is limited by its encoding speed. Since this encoding is inefficient for dynamic content, the maximum frame rate that can be reached is 20 fps for a relatively small video area, and for larger motion areas the frame rate drops under 10 fps. With the video streaming encoding, the frame rate is above 24 fps, independent of the encoded content. A slightly increased frame rate is reached for smaller motion areas, since the similarity of the static areas of the screen can be encoded faster than large motion areas that need somewhat heavier encoding. Perfect detection combines best of both worlds, providing over 24 fps for the motion area, while the static area updates are transmitted upon user input.

However, the achieved frame rate is only one of the two factors that determine the visual quality experienced by the user. The second metric concerns the similarity between the frames delivered at the viewer side and the original rendered content at the server side. VNC divides screen updates into rectangles, with optimized encodings for e.g. text, gradients and lines, to encode these rectangles efficiently in high quality. Video streaming codecs encode motion content well, but their lossy, motion vector based encoding makes them less appropriate for static content that most often consists of text and lines. Table 2 depicts the similarity between server-side rendered text displayed with gedit text editor and the resulting delivered update at the thin client viewer side, for both video encoding and VNC encoding. This similarity is evaluated using two metrics commonly used in the field of video coding: the Structural Similarity (SSIM) score [23] and the Peak Signal-to-Noise Ratio (PSNR). The PSNR of an image frame represents the pixel-per-pixel deviation (measured as the Mean Squared Error (MSE)) from the original content. Although this is an objective metric (not capturing subjective effects), it is frequently used in the field of video encoding to assess Quality of Experience (QoE) [24][1]. Both metrics indicate that VNC is able to represent text very well. The small divergence from 100% similarity is caused by the viewer-side rendered mouse overlay which does not appear in the server side frame buffer capture. In contrast, when applying the video coding for the static text content the representation improves gradually, starting from a blurred, unreadable text to a slightly blurred, readable text. This effect is clearly visible for the end user, as the correct representation of the text is shown by the thin client viewer after more than 30 frames (corresponding to over 1 second). In the table the average values over the 30 frames are reported,

Table 2: Comparison of degradation caused by video streaming and by VNC encoding, using Peak Signal to Noise Ratio (PSNR) and Structural Similarity (SSIM) metrics.

Content	Encoding	PSNR	SSIM
Static text	H.264	35.05 dB	0.9734
	VNC	39.28 dB	0.9979
Video	H.264	62.88 dB	0.9989

[1]Note that both PSNR and SSIM metrics require a reference for quality assessment. In contrast to purely video coding scenarios, in thin client computing, this reference is not always easy to acquire. For these tests, we specifically had the reference of a static scene. However, for dynamic content the acquisition of reference material is more complex due to the differing frame rate delivered compared to the original. In this case, there is also no consensus on which metric value to take for frames that do not occur in the degraded sequence. Active research towards no-reference subjective quality metrics is currently conducted, but for now none of those correlate well enough to the subjective standardized metrics [25].

while the PSNR increases from 25.4 dB to 38.5 dB, and the SSIM from 0.85 to 0.99.

5.3. Algorithm Sequence Parameter Settings

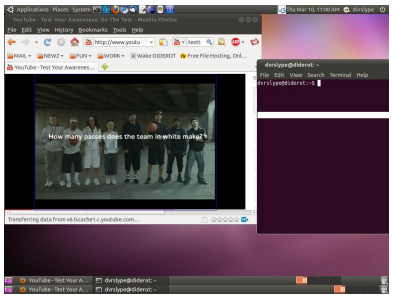
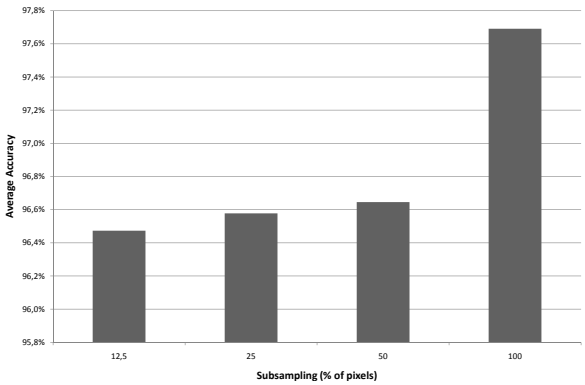


Fig. 7: Screenshot of the application layout in the experiments, displaying a YouTube movie and a Linux command shell window.

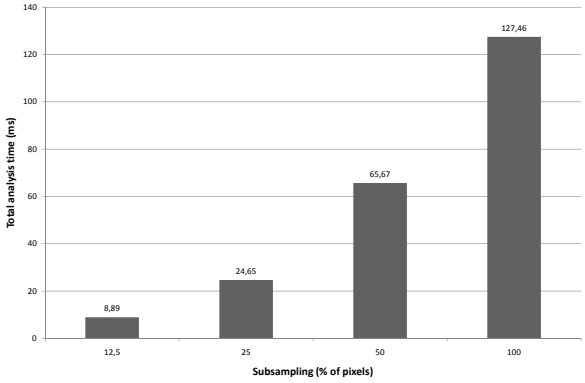
Parameter settings were optimized using the following scenario: a session with two active windows is displayed on the screen. One window plays a YouTube movie[2], while the other windows displays a Linux command shell window, as shown in Fig. 7. This scenario was controlled so that the exact position and size of the video was known, enabling correct assessment of the accuracy (expressed as the number of pixels that were incorrectly classified) of the analysis. The exact same sequence was processed with both the pixel analysis and the VNC rectangle analysis algorithm. The outputs of both are compared to the known positions of the motion regions, allowing to assess the accuracy of the system. Apart from evaluating the accuracy, we have also measured the processing time required to perform the analysis, indicating the feasibility for real-time processing and hinting of motion regions. These experiments were performed for a desktop with a resolution of 1024×768.

5.3.1. Sampling

Decreasing the number of samples that are taken into account has a positive effect on processing speed of the complete analysis, but will limit the accuracy of the derived hints, as can be seen in Fig. 8(a). If all pixels of the screen are included to find motion region, the average accuracy in our test scenario is 97.69%. Sampling only 12.5% of the pixels degrades the accuracy by 1.22%. The effect of sampling on the complete pixel analysis time is depicted in Fig. 8(b). As expected from the complexity analysis presented in Section 4.1, this analysis time is linearly dependent of the number of samples taken. For sampling 100% of the pixels, the analysis takes 127.46 ms. Sampling 12.5% of the pixels decreases that time to 8.89 ms.



(a) Variation of sampling raster sizes, influence on pixel analysis accuracy.



(b) Variation of sampling raster sizes, influence on analysis time.

Fig. 8: Influence of sampling raster sizes.

In addition to the number of samples taken per frame, also the analysis frequency has an impact on the computational requirements. More specifically, both parameters influence the amount of memory used to store a time window of screen captures. While the upper bound of the sampling frequency is limited by the capacities of the server, the lower bound for the sampling frequency is defined by the threshold for classifying dynamic parts over static ones. E.g. when the threshold to categorize content as motion is set to 15 changes per second, at least that amount of samples needs to be taken per second to be able to detect motion content. For our experiments, the thresholds and

[2] Awareness test, http://www.youtube.com/watch?v=oSQJP40PcGI

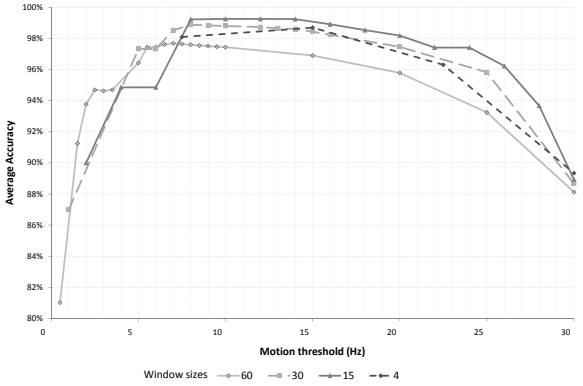
window sizes were varied, while keeping the sampling frequency fixed at 30 Hz.

5.3.2. Sliding Time Window Size

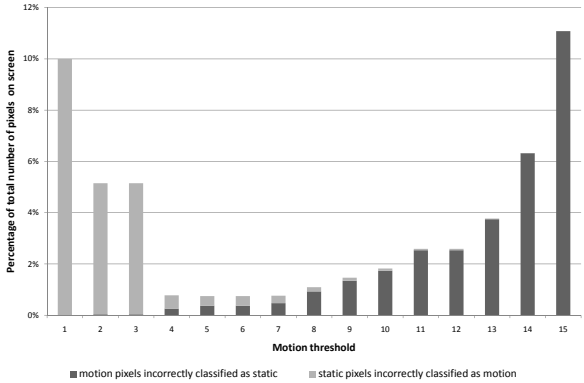
Figure 9 shows that the sliding time window size, denoted as w in Listing 1 and Listing 2, affects the reaction speed of the system. This reaction speed could be seen as the time between a change in content type on screen, and the analysis detecting this change. On the other hand, the time window size also mitigates small temporary variations in content type. In the experiment, the YouTube video started playing after 60 frames. At frame number 338, the video paused and subsequently resumed at frame 506. The figure depicts 4 different window sizes, with a motion threshold (referred to as *motionThreshold* in Listing 1 and Listing 2) set to 10 changes per second, which at the configured sample frequency of 30 Hz amounts to a motion threshold of 20 for a window size of 60 frames, motion threshold 10 for window size 30, and motion threshold 5 for window size 15. This motion threshold setting of 10 changes per second was determined from experimentation with the test scenario, and represents the point with the best accuracy from the training set, as presented in Fig. 10(a). Considering that an average typist reaches about 200 keystrokes per minute (or somewhat less than 4 per second) and video files are conveniently encoded above 15 frames per second, a value in between as motion threshold is reasonable. When the window size is chosen too large, e.g. 60 frames, the system reacts very slow to changes. Quick variations in content type are filtered out, e.g. during the "video pause" part, while detection of actual changes of content type is delayed. On the other hand, configuring the window size too small, e.g. 4 frames, the system reacts fast to actual changes but exhibits some classification errors as shown in the "video pause" part. Compared to the ideal sliding window size (i.e. 15 frames), both deviations lead to longer periods of non-optimal encoding and visual artifacts. This is also clear from Fig. 10(a), where this window size clearly exhibits the highest accuracy values.

5.3.3. Motion Threshold

Figure 10(a) presents the average accuracy of the complete session, for varying motion thresholds with multiple window sizes. As explained earlier, the window size of 15 frames leads to the maximum accuracy due to an adequate reaction speed. From the results in the figure, we derived an optimal configuration of motion threshold 5 for a window size of 15 frames (amounting to 10 changes per second). The curves in the figure show that deviations from this optimum lead to decreased accuracy. This can be clearly explained using Fig. 10(b). For low motion thresholds, samples are considered to be part of motion content when they change relatively infrequent. Thus choosing the motion threshold too low increases the share of static pixels incorrectly classified as motion. Likewise, for high motion thresholds, samples are considered to be part of motion content when they change very frequently. Configuring the motion threshold too high increases the share of motion pixels incorrectly classified as static.



(a) Influence of motion threshold on pixel analysis accuracy.



(b) Influence of motion threshold on incorrect classification for a window size of 15 frames.

Fig. 10: Variation of motion threshold. Remark that in figure (b), the window size of 15 frames implies that a motion threshold of 5 leads to a minimum classification errors. This value corresponds to a motion threshold of 10 Hz as used as the unit in figure (a), for which the maximum accuracy can be read for window size 15.

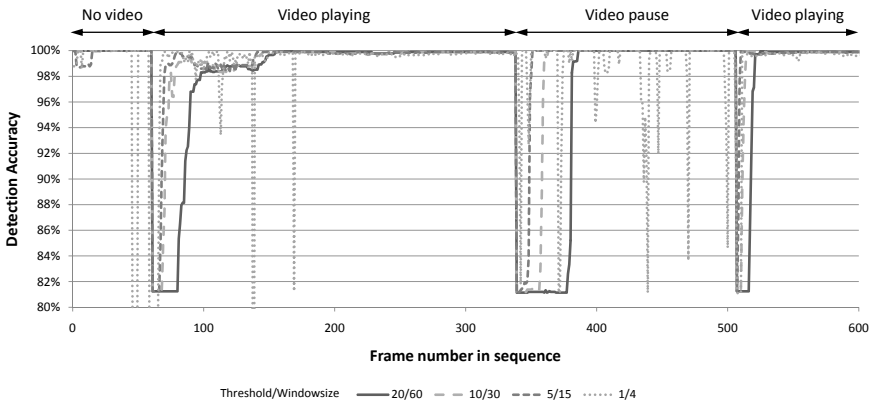


Fig. 9: Variation of sliding window sizes, influence on pixel analysis accuracy.

5.3.4. Blob Extraction

For blob extraction, the minimum size and the form of the blobs can be configured. For all experiments, the IPL98 library was configured to detect blobs of minimum size width and height of 64 pixels, as such small regions are considered to be efficiently encodable with VNC encoding. The hybrid video encoder requires the specification of rectangular zones (that will be encoded using either a thin client protocol or video encoding), hence the need to convert the blobs to rectangles. Given the support from the video encoder, a rectangular form for the blobs is requested. The blob library was configured to use the 8-connected variant of the Connected Component Labeling (CCL) algorithm. However, the library does not implement the linear time optimization described in [21].

Experimentation with the blob sizes teached us that configuring the blob sizes very small results in decreased accuracy due to small but frequent changes that are classified as a motion region. However, such small regions do not require specific video encoding as it can be handled well by the VNC encoding as well. More interestingly, we have found that, for a controlled test with a video region of known size and blob sizes above 50 by 50 pixels, the accuracy remains unaffected by the blob size until it is configured to be greater than the actual video itself.

5.3.5. VNC rectangle parameter settings

The same scenario as before (Fig. 7) was used to assess the VNC rectangle approach, investigating the motion threshold and time window size parameters. The results are presented in Fig. 11. Overall, the accuracy reaches a maximum of 90.56%. This optimum is reached using motion threshold 26 for a time window size of 30 frames. In comparison to the pixel analysis, this is a relatively high threshold. Inspection of the analysis output from the session depicted in Fig. 12 shows that the hints delivered with these settings have a large variance which is rather difficult to be used as input for the thin client encoder.

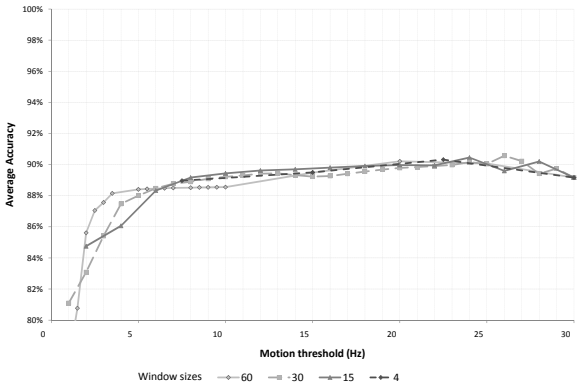


Fig. 11: Influence of motion threshold on VNC rectangle analysis accuracy.

Figure 13 presents the accuracy of the VNC rectangle analysis compared with that of the pixel analysis, for identical settings i.e. motion threshold 5 for time window size 15. The results show that the accuracy of the rectangle analysis is lower than that of the pixel analysis. This can be attributed to padding for scaling the video according to its aspect ratio, and the pauze in the video which are filtered out by the pixel analysis.

The difference with pixel analysis is that even video

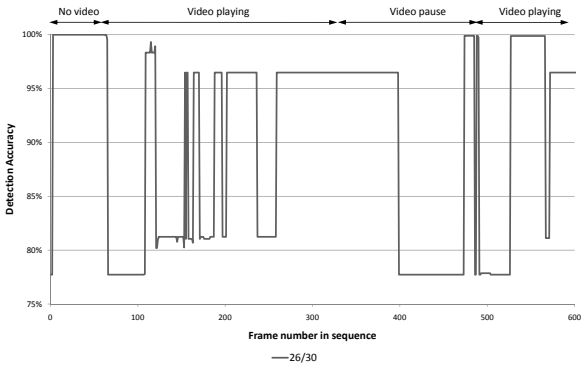


Fig. 12: Classification per frame in sequence for VNC rectangle analysis with motion threshold 26 for a window size of 30 frames.

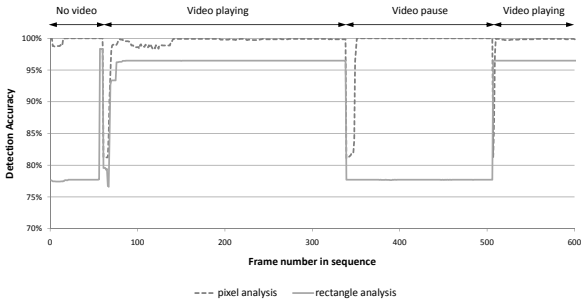


Fig. 13: Comparison of classification per frame in sequence for VNC rectangle analysis and pixel analysis with motion threshold 5 for time window size 15 frames.

files that contain static scenes are still transported as video, causing a significant bandwidth increase. However, it is clear that the thin client protocol rectangle analysis tends to overestimate the dynamic regions in the screen (as explained before in Section 4.3). This knowledge can be used to improve the performance of the pixel analysis. Indeed, we could combine both approaches as follows: use rectangle analysis to decide on the static regions, and refine the classifications of "dynamic" regions by pixel analysis. This limits the more complex pixel analysis to a subset of the entire screen.

5.4. Analysis time evaluation

For both implemented algorithms, we have separately recorded the time to compute all steps. For a proof-of-concept software implementation, the results are presented in Table 3, showing that sampling is a good strategy to decrease the time needed for the complete pixel analysis, as it scales nearly linearly with the number of samples taken. This is also clear from the order of complexity as expected based on the analysis presented in Section 4.1. For blob detection, the execution time is more than halved when halving the number of samples. This non-linear behavior is due to the IPL library that does not implement the linear time CCL algorithm optimization. However, compared to difference frame computation and thresholding, the blob detection is a minor piece of the total execution time. For the rectangle analysis, the difference frame computation is not required as the modified regions are used as input for the algorithm. The maintenance of the time window happens faster than for pixel analysis because it is executed in the rectangle domain. For thresholding and blob detection, the analysis needs to be done at pixel level, causing the total analysis time to add up to 11.368 ms. Remark that this should be compared to 100% sampling and is 10 times faster than pixel analysis in this case. Thus, the rectangle analysis can be optimized for computation speed by either sampling when converting between the pixel and the rectangle domain, or by performing the entire analysis, i.e. including the thresholding and blob detection step, in the rectangle domain.

5.5. Validation

To validate the parameters found in Section 5.3, this configuration was applied to two new scenarios. With the first scenario, the accuracy of the detection system is challenged against motion content with different characteristics. The second scenario against which we validate our algorithms represents a real life scenario with user actions such as typing text, scrolling through a text, watching pictures and moving windows around. The same hardware setup as described in Section 5.1 was used to perform both experiments.

5.5.1. Four movies validation scenario

The first scenario consists of four videos with different characteristics, that were played back one at the time on different positions on screen. The first video that was played is a computer generated animation movie that can be catalogued as visually dynamic. The second sequence is an action scene from a visually dynamic natural movie. The third movie in the sequence is a rather visually static natural movie. The last shown video is a visually static computer generated animation

Table 3: Time analysis of detection algorithms.

	sampling and diff. frame computation*	windowing†	thresholding‡	blob detection$^\circ$	total
Pixel analysis					
100% sampling	50.834 ms	8.610 ms	52.502 ms	15.517 ms	127.463 ms
50% sampling	28.048 ms	4.330 ms	28.809 ms	4.477 ms	65.665 ms
25% sampling	9.993 ms	2.245 ms	12.097 ms	0.317 ms	24.651 ms
12.5% sampling	5.465 ms	1.181 ms	2.153 ms	0.093 ms	8.892 ms
Rectangle analysis	n/a	1.189 ms	4.630 ms	5.368 ms	11.368 ms

* downsampling an input frame and computing the difference with the previous input frame. For rectangle analysis this step is not needed, hence the non applicable (n/a) value.

† creating and maintaining the time window over which the motion values are computed.

‡ computing the binarized motion frame that is acquired through the threshold step of the algorithms.

$^\circ$ extracting blobs in the binarized motion frame.

movie[3]. The times at which the videos were visible on screen are indicated in the figures reporting results obtained with this scenario. The screen resolution of the thin client session was 1024×768.

Figure 14 presents the results of the first validation experiment. Figure 14(a) shows the accuracy with pixel analysis for both 100% and 12.5% of the pixels used as samples for the analysis. Apparently, for the settings used, i.e. motion threshold 5 for a window size of 15 frames, the subsampling has a very small effect on accuracy. The pixel analysis shows good performance for dynamic content and is able to detect relatively static natural moving content very well, but shows lower accuracy for detecting low dynamic computer generated content. Figure 14(b) compares the accuracy of the pixel analysis with the VNC rectangle analysis and shows that the VNC rectangle analysis is able to detect the low dynamic computer generated content well, but has the clear tendency to overestimate motion regions. This is the case for both natural movie parts in the scenario, where the padding for correct resolution scaling is redrawn with the video frames, transparently for the VNC analysis (as explained in Section 4.3). This overestimation is visible in Fig. 14(c), where — except for during content changes — the share of incorrect classifications is entirely caused by static pixels that are classified as motion content. In summary, the average accuracy achieved with both pixel analysis and VNC rectangle analysis for the four different content types in the experiment is reported in Table 4.

Table 4: Average accuracy depending on content type.

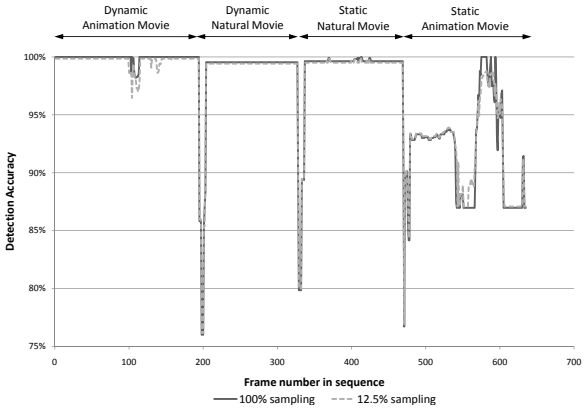
	pixels (sampling)		rectangles
	100%	12.5%	
Dyn. Animation	99.93%	99.67%	100%
Dynamic Natural	99.54%	99.42%	96.49%
Static Natural	99.58%	99.46%	96.50%
Static Animation	92.03%	92.21%	99.92%

5.5.2. High Definition validation scenario

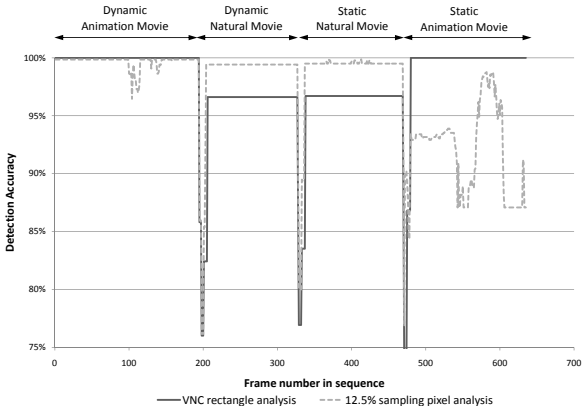
In the second validation scenario, the desktop session was performed in High Definition (Full HD, 1920×1080 pixels) resolution. A series of actions that occur frequently in a normal day-to-day computer usage cycle was performed: opening and closing applications, editing text, scrolling through a large document, watching a series of pictures, rotating a picture, dragging a window over the screen and minimizing and maximizing a window. The same settings were applied as identified appropriate before, i.e. motion threshold of 5, window size of 15 frames and 12.5% sampling.

As expected for *opening and closing applications*, *editing text*, *watching a series of pictures*, *rotating an image*, and *minimizing and maximizing an application*, our algorithms classified the screen entirely as static content. For these actions, the changes do not occur frequently enough or are too small to be identified as a dynamic region worth video streaming. For *scrolling through a large document*, an all-static classification was obtained. Although such action might be expected

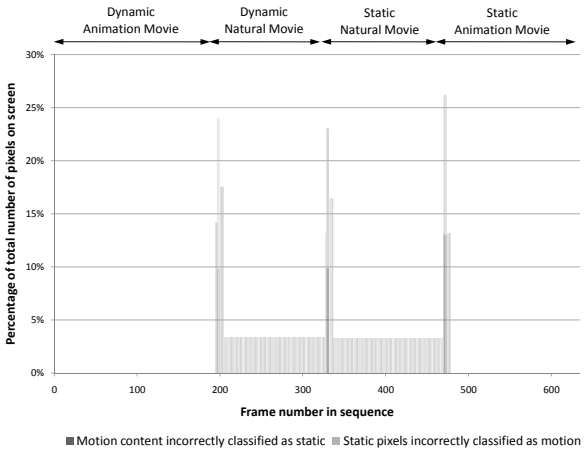
[3]The respective videos that were used are "Sintel" (http://www.sintel.org), a trailer from "Spiderman", a trailer from "Das Leben Der Anderen" and finally "Big Buck Bunny" (http://www.bigbuckbunny.org).



(a) Pixel analysis.



(b) VNC rectangle analysis.



(c) VNC rectangle analysis, share of incorrect classifications.

Fig. 14: Validation scenario with 4 videos.

to receive a high motion classification at first sight, the resulting window matrices consist of scattered changes that are too small to be detected as motion areas. Indeed, the content alters greatly during scrolling, but given the more textual nature, the difference frames are constituted merely of coincidental similarities and dissimilarities.

The most challenging part of the experiment occurred when *dragging a window over the screen*. A video was played back in windowed mode (i.e. non full screen) and was detected as a motion area with a similar accuracy as in the previous experiments. Then, while playing the movie, this window is dragged over the screen. At its new position, the video region was classified as motion accordingly. However, during the dragging itself, classification accuracy is decreased. Namely, in the window matrix, the locations the motion region moves away from, and new locations of the motion region, respectively fade out and fade in slower than actually occurring on screen. An intersection of the dragged content over the time window is detected as motion. Such phenomena are easily detectable, and could be accounted for by excluding the derived classifications as hints to the encoder until they are stabilized or residing to a particular solution for such scenarios. For example, a particular solution could be to encode the entire screen as VNC encoding, since the contents of dragged windows typically are less of interest to the user until it gets in its proper place, although further investigation would be needed to conclude this formally. Repeated experiments with the dragging of a window with a single image (i.e. non-motion content) lead to the same conclusion.

The computation times recorded for this High Definition resolution scenario are presented in Table 5. The total time needed to analyse one frame using 12.5% sampled pixel analysis results to 15.4 ms with the proof-of-concept software implementation. So also for Full HD resolutions, real-time analysis is feasible.

5.6. Interpretation of experimental results

Concerning the interpretation of the accuracy metric used in this paper, the location of wrongly classified pixels has been analysed for the scenario used in Fig. 7, with a motion threshold of 5 and a window size of 15 frames. Figure 15 presents the impact of misclassification, expressed as the deviations from the boundaries of the hints with respect to the actual dynamic region. Negative values mean that the dynamic region has been underestimated, positive values indicate overestimations. E.g. a value of -10 means that the border of a dynamic region has been estimated to lay 10 pixels within the

Table 5: Time analysis of detection algorithms for Full HD validation scenario.

	sampling and diff. frame computation	windowing	thresholding	blob detection	total
Pixel analysis 12.5% sampling	6.543 ms	2.890 ms	3.477 ms	2.463 ms	15.373 ms

actual region (+10 implies a border position 10 pixels outside the actual region). In the case that all pixels are evaluated, i.e. 100% sampling shown in Fig. 15(a), the majority of misclassifications is due to a hint that deviates only by 1 to 2 pixels. The maximum recorded misclassification amounts to 10 pixels underestimation of the dynamic content, and only occurs in less than 10% of the errors. When 12.5% of the pixels are sampled, boundary errors are distributed differently because of rescaling, as presented in Fig. 15(b). These results show that estimation errors manifest themselves primarily at the borders of the regions. Therefore, the impact for the user is expected to be limited, as for video regions in most cases the interesting information is more or less centered. More generally, we could pose that cases where the boundaries contain critical information for which optimal encoding is crucial are rather rare, although this needs to be investigated more thoroughly.

In this section, a variety of parameters has been investigated with respect to their impact on the performance of the proposed algorithms. The main conclusions and trade-offs are summarized below:

- Sampling: In the case of pixel analysis, sampling has a major effect on the analysis speed, as it has a direct influence on the number of pixels and consequently the amount of computations that need to be performed. However, this speed-up comes at the cost of decreased accuracy of the algorithm output. The computational complexity scales linear with the number of pixels analysed. Our proof-of-concept software implementation shows to be able to process Full HD content in real-time under 12.5% sampling, i.e. 15.4 ms are needed per iteration of the algorithm.

- Sliding window size: the size of the sliding window determines the reaction speed of the analysis component. Choosing a large sliding window makes the system react slowly to effective changes in the session, avoiding unnecessarily fast switches. Conversely, small window sizes result in a system that detects changes faster but is also susceptible to hasty classifications of certain content types. From our experiments, we found that configuring a sliding window size of 15 frames at a sampling rate of 30 Hz, i.e. a window size of 0.5 s, gave the best trade-off between reaction speed and accuracy.

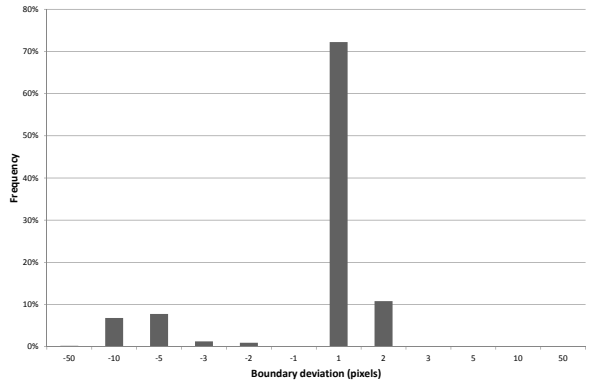
- Motion threshold: the motion threshold determines the actual differentiation between motion versus static content. The actual setting of the motion threshold depends on the sliding window size and the sampling rate, in that an absolute motion threshold of X Hz at a sampling rate of Z Hz results in a configuring motion threshold of $X \times \frac{Y}{Z}$ frames for a window size of Y frames. In our tests, a motion threshold of 10 Hz, i.e. 5 frames for a window size of 15 frames at 30 Hz sampling rate, resulted in the highest accuracy for the proposed algorithms.

- Blob sizes: the blob size can be determined as the minimum size of motion regions that are to be detected. All motion regions smaller than the blob size will be filtered out and considered static after blob detection. The setting used for the experiments in this article was a blob size of 64 by 64 pixels, and gave satisfactory results.

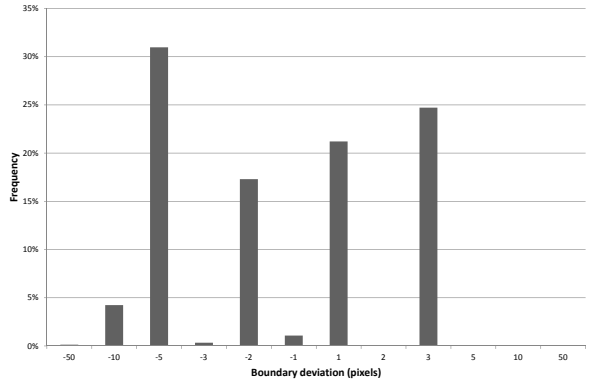
Misclassification effects: classification errors are concentrated around the boundaries of detected regions. The majority of the misclassifications result in 1 to 2 pixels incorrectly detected, and a maximum deviation of 10 pixels off was recorded.

6. Conclusions

This article proposes algorithms for automatic detection of different content types for encoding the visual content for thin client computing. The approach separates the content type detection and thin client protocol encoding, in order to enable them to perform at different speeds and to allow for hysteresis in the detection component so that hints given to the encoder can be stabilized. The need for a detection system is shown via



(a) Histogram of boundary deviations for 100% sampling.



(b) Histogram of boundary deviations for 12.5% sampling.

Fig. 15: Histograms of boundary deviations, indicating the effect of misclassification. The deviations represent the differences between the four boundaries (upper, lower, left and right) of the hints and those of the actual dynamic region. These results relate to the scenario described in Section 5.3, using motion threshold 5 and window size of 15 frames.

experimental results on both consumed bandwidth and user experience, and two algorithms are discussed to perform such detection, i.e. pixel analysis and thin client protocol rectangle analysis. The detection accuracy of both algorithms is evaluated, as well as the performance in terms of computation speed.

Overall, we conclude that both levels of analysis have their own advantages. Thin client protocol rectangle analysis can work faster than pixel analysis because this algorithm operates in the rectangle representation domain, which allows optimized mathematics and aggregated evaluation of areas on screen. Pixel analysis is computationally more intensive (although this can be greatly alleviated by performing subsampling), but is able to overcome overestimation issues specific to rectangle analysis, leading to higher accuracy. For low motion, computer generated dynamic content, the rectangle analysis outperforms pixel analysis. Given these findings, a combined approach is expected to be beneficial for both analysis speed and accuracy. The thin client protocol rectangle analysis could perform a rapid coarse grained scan to be used as input for the pixel analysis, leading to focused sampling and fine grained area detection. Experiments with prototypes of both algorithms result in accuracies between 92% and 100%, within 8.892 ms with the pixel analysis algorithm and 11.368 ms with the rectangle analysis algorithm. Also for Full HD content, the proof-of-concept software implementations of the proposed algorithms show to be viable for real-time analysis of the desktop session.

Future work includes automatic deduction of optimal parameter settings for the detection algorithms, the detection of additional content types besides dynamic and static, and feedback of user perceived quality to guide the hinting algorithms. It would also be interesting to study the decoupling and synchronization between analysis and encoding components in more detail. Furthermore, we expect adaptive thresholds to build in intelligent inertia for switching between modes to be beneficial. Also the solution proposed for dragging windows over the screen, i.e. on detection of this case, the system should reside to full screen VNC encoding, should be investigated more thoroughly.

Acknowledgement

This work has been partly funded by the European Community's Seventh Framework (FP7/2007-2013) under grant agreement no. 216946, in the scope of the MobiThin project and by the Research Foundation – Flanders (FWO – Vlaanderen) in the scope of the Thin Client Network Management project (G.039107N). Vankeirsbilck is funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Develder is supported in part as a post-doctoral fellow of FWO – Vlaanderen.

References

[1] T. Petrovic, K. Fertalj, Demystifying desktop virtualization, in: 9th WSEAS International Conference on Applied Computer Science (ACS'09), Recent Advances in Computer Engineering, WSEAS, Genoa, Italy, 2009, pp. 241–246.

[2] Microsoft Corporation, Description of the Remote Desktop Connection 7.0 client update for Remote Desktop Services (RDS) for Windows XP SP3, Windows Vista SP1, and Windows Vista SP2, http://support.microsoft.com/kb/969084.

[3] Citrix Systems, Inc., HDX(TM) Media Stream, High Definition User Experience, http://hdx.citrix.com/hdx-mediastream.

[4] P. Simoens, P. Praet, B. Vankeirsbilck, J. De Wachter, L. Deboosere, F. De Turck, B. Dhoedt, P. Demeester, Design and implementation of a hybrid remote display protocol to optimize multimedia experience on thin client devices, in: Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian, 2008, pp. 391–396. doi:10.1109/ATNAC.2008.4783356.

[5] J. Nieh, S. J. Yang, N. Novik, Measuring Thin-Client Performance using Slow-Motion Benchmarking, ACM Trans. Comput. Syst. 21 (1) (2003) 87–115. doi:10.1145/592637.592640.

[6] A. M. Lai, J. Nieh, On the performance of wide-area thin-client computing, ACM Trans. Comput. Syst. 24 (2) (2006) 175–209. doi:10.1145/1132026.1132029.

[7] A. Lai, J. Nieh, On the performance of wide-area thin-client computing, ACM Trans. Comput. Syst. 24 (2) (2006) 175–209. doi:10.1145/1132026.1132029.

[8] T. Richardson, Q. Stafford-Fraser, K. R. Wood, A. Hopper, Virtual network computing, IEEE Internet Computing 02 (1) (1998) 33–38. doi:10.1109/4236.656066.

[9] NoMachine, FreeNX, http://freenx.berlios.de/.

[10] Microsoft Corporation, Remote desktop protocol performance improvements in windows server 2008 r2 and windows 7, Tech. rep., Microsoft (January 2010).

[11] Red Hat, Spice protocol, http://www.spice-space.org/.

[12] Teradici Corporation, PC-over-IP remote display technology: true zero client desktop virtualization - PCoIP, http://www.teradici.com/.

[13] K.-J. Tan, J.-W. Gong, B.-T. Wu, D.-C. Chang, H.-Y. Li, Y.-M. Hsiao, Y.-C. Chen, S.-W. Lo, Y.-S. Chu, J.-I. Guo, A remote thin client system for real time multimedia streaming over vnc, in: Multimedia and Expo (ICME), 2010 IEEE International Conference on, 2010, pp. 992–997. doi:10.1109/ICME.2010.5582993.

[14] ISO/IEC, MPEG 4 Standard JTC1 SC29 WG11 (2002).

[15] Y. Lu, S. Li, H. Shen, Virtualized screen: A third element for cloud-mobile convergence, Multimedia, IEEE 18 (2) (2011) 4 –11. doi:10.1109/MMUL.2011.33.

[16] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, C. Bouras, Platform for distributed 3d gaming, Int. J. Comput. Games Technol. 2009 (2009) 1:1–1:15. doi:10.1155/2009/231863.

[17] A. Boukerche, R. W. Pazzi, J. Feng, An end-to-end virtual environment streaming technique for thin mobile devices over heterogeneous networks, Computer Communications 31 (11) (2008) 2716 – 2725, end-to-End Support over Heterogeneous Wired-Wireless Networks. doi:10.1016/j.comcom.2008.02.032.

[18] B. Vankeirsbilck, P. Simoens, J. De Wachter, L. Deboosere, F. De Turck, B. Dhoedt, P. Demeester, Bandwidth optimization for mobile thin client computing through graphical update caching, in: Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian, 2008, pp. 385 –390. doi:10.1109/ATNAC.2008.4783355.

[19] W. Shi, Y. Lu, Z. Li, J. Engelsma, SHARC: A scalable 3d graphics virtual appliance delivery framework in cloud, Journal of Network and Computer Applications 34 (4) (2011) 1078 – 1087, advanced Topics in Cloud Computing. doi:DOI: 10.1016/j.jnca.2010.06.005.

[20] R. Walczyk, A. Armitage, D. Binnie, Comparative study on connected component labeling algorithms for embedded video processing systems., in: H. Deligiannidis (Ed.), IPCV'10, Vol. 2, CSREA Press, Las Vegas, USA, 2010.

[21] F. Chang, C.-J. Chen, C.-J. Lu, A linear-time component-labeling algorithm using contour tracing technique, Computer Vision and Image Understanding 93 (2) (2004) 206 – 220. doi:10.1016/j.cviu.2003.09.002.

[22] K. Suzuki, I. Horiba, N. Sugie, Linear-time connected-component labeling based on sequential local operations, Computer Vision and Image Understanding 89 (1) (2003) 1 – 23. doi:10.1016/S1077-3142(02)00030-9.

[23] Z. Wang, A. Bovik, H. Sheikh, E. Simoncelli, Image quality assessment: From error visibility to structural similarity, IEEE Transactions on Image Processing 13 (4) (2004) 600–612. doi:10.1109/TIP.2003.819861.

[24] S. Winkler, P. Mohandas, The evolution of video quality measurement: From psnr to hybrid metrics, Broadcasting, IEEE Transactions on 54 (3) (2008) 660 –668. doi:10.1109/TBC.2008.2000733.

[25] Video Quality Experts Group, Report on the validation of video quality models for high definition video content, Tech. rep., http://www.vqeg.org/ (2010).