An Open Source Architecture for Low-Latency Video Streaming on PDAs

Giovanni Gualdi[a], Andrea Prati[b], Rita Cucchiara[a]
{gualdi.giovanni, prati.andrea, cucchiara.rita}@unimore.it
[a]DII - [b]DISMI, University of Modena and Reggio Emilia, Italy

Abstract

This paper presents a open-source system for low-latency video streaming on PDAs, specifically addressing mobile video surveillance requirements. The system is based on H.264 and suitably modified to obtain the best trade-off between image quality and video fluidity, working also at very limited bandwidths. Moreover, the used controls allow to keep the number of lost frames very low. A large set of experiments and comparisons have been carried out and the achieved results demonstrate the efficacy and efficiency of our system.

1. Introduction

The fruition of multimedia content, especially videos, from every place is nowadays mandatory for several different scenarios: people want to watch entertainment videos on the portable DVD player, to access to digital live broadcasting of news or sport events by using the cellular phone or to watch at a surveillance camera pointing to the office, the factory or the house by using a PDA. Due to the advances in the technology and the increased need of mobility for people, *Ubiquitous Multimedia Access* to access to video digital libraries and to live videos is mandatory. Our target scenario is *mobile video surveillance*, that is the real-time and ubiquitous availability of security-related video, directly acquired by surveillance cameras. A typical example is a security officer or a policeman monitoring a specific remote site looking at a live camera through his PDA.

This type of application presents several challenges from the technological point of view. First, video data requires either wide bandwidth for being remotely transmitted or complex (thus, computationally expensive) compression algorithms: the goal is to fit limited bandwidth and preserve, at the same time, the most of the video quality. Second, the accessibility through wireless networks, either 802.11 WiFi, 3G (UMTS), or even 2G or 2.5G (GPRS) networks, is required. In our application both the transmitter and the receiver are supposed to be free to move where WiFi coverage is not guaranteed. For this reason, we have concentrated on radio mobile networks that have wide territorial coverage (at least over Europe), though very limited capacity. Specifically we relied on GPRS and E-GPRS (GPRS-EDGE), both GSM based, intentionally avoiding UMTS networks that still today in Europe offer a reliable network coverage only in urban areas.

In our previous work [5], we have described a system for low-latency video streaming on low-capacity networks, addressing PC-to-PC architecture, GPRS/E-GPRS communications and high degree of mobility, either on the encoder or on the decoder. In the current paper we extend these capabilities by developing a complete open-source tool for video streaming from PC to mobile device (specifically a PDA). The novelties with respect to our previous work are due to the additional constraints imposed by the fact that the decoding of the streamed video needs to be performed on a PDA or, in general, on a device with limited resources. More specifically, besides the display limitation (that can affect user satisfaction but is not so relevant in our case, being the video already scaled to fit low bandwidth requirements), PDA devices present also the following limitations:

- *limited computational resources*: processors on board of PDAs are becoming more and more powerful but are still insufficient for intensive real-time computations;
- *no reference architecture*: a strongly affirmed reference architecture for PDAs does not exist yet;
- *embedded operating system*: PDAs often have special (limited) operating systems that require ad-hoc solutions and applications.

Talking about mobile processing means to deal with the problem of energy consumption. There is a whole branch of ongoing research on this topic but we will not address it in the present work because out of scope of the paper.

The target application we envision presents certain requirements (partially common to those in [5]) that must be considered in choosing the technological and algorithmic best solution:

1. *live video delivery* requires efficient video coding and transmission;

2. since video surveillance requires a *good understanding of the scene*, the received stream needs to be of both high quality of images and good fluidity of video, related to the decoding framerate; it is important to underline that these two requirements are often in conflict, given the strict bandwidth limitations;

3. since video surveillance requires often *quick reaction to the events* (to allow interaction with the remotely observed scene), a low latency in the whole streaming process is required; the usual latency introduced by entertainment live video streaming (usually more than 5 seconds), is not acceptable in surveillance scenarios;

4. the system must be based on *open-source software*.

Being restricted to use low-capacity networks like GPRS, compression techniques such as MPEG-2 and MPEG-4 are not sufficient. The evolution of MPEG-4 and H.263, called H.264/AVC [1], offers extremely good video quality even at very high compression rates: this makes it very suitable for the scope of our project. As a drawback it presents high computational complexity (both on encoder and decoder side); however, if the encoder is fed with reasonable resolution video streams and properly-tuned compression parameters, it is possible to use H.264 on real-time systems [11]. The reason for requirement #4 (open-source software) is twofold. First, the availability of open-source software for different platforms allows to cross-compile it on different architectures and/or operating systems, including the embedded operating system of the PDAs. Second, the availability of the complete source code permits modifications and optimization at any level.

2. Analysis of the state of the art

Existing commercial or off-the-shelf streaming solutions for live video streaming, decompression and playback on PDAs are not sufficient to meet our requirements; to verify this, we have performed thorough tests on some of them (widely and analytically described in the experimental results). Table 1 summarizes the main pros and cons of the compared tools.

The most diffused software tools for video streaming (i.e, Microsoft Windows Media® suite, Real Networks® suite - based on Helix technology - or Quicktime® suite) have been developed more for entertainment purposes and for handling intensive video broadcasts, rather than for ubiquitous live video streaming, and thus introduce high latency, that does not meet our requirement #3. In particular, Microsoft Windows Media Encoder® reduces drastically the framerate when the available bandwidth is less than 15 kbps (see experimental results) and cannot handle compression with bitrate lower than 7 kbps.

From the scientific point of view, a survey paper on video streaming has been reported by Lu in 2000 [9]. It reports of and discuss about the relevant signal processing issues related to video streaming and proposed possible solutions. In general, regarding video streaming, researchers have addressed the problem from several different perspectives. For instance, Lu *et al.*in [10] basically considered the allocation of computational resources for a server that encodes live videos and receives more clients with different priorities in accessing to the videos. Instead, works such as [4] and [6] mainly addressed video streaming in terms of system architecture. In particular, Conklin *et al.*in [4] proposed several models of communications (based on RealNetworks® products) for video delivery on the Internet, not considering low-capacity networks. Guo *et al.*in [6] proposed a very nice application for live video streaming between moving vehicles and depicted an interesting architecture for data synchronization. However, their system is based on 802.11 WiFi networks and thus it is not suitable to our case.

Some previous works have proposed systems for video streaming over low-capacity networks, such as GPRS. For instance, Lim *et al.*in [7] introduced a PDA-based live video streaming system on GPRS network. The system is based on MPEG-4 compression and contains several computational optimizations for working on a PDA. It can achieve a frame rate of 21 fps (frames per second) at the encoder side and 29 fps at the decoder side for transmitting at 128 kbps a video in QCIF (176x144) format. However, their system drops to 2-3 fps when transmission is over GPRS. This limitation is basically due to the use of MPEG-4. Moreover, no information on the latency of the system is provided. The work in [8], instead, solves the problem of transmitting real-time videos over GPRS by using frame skipping. Unfortunately, in our target application, frame skipping is not suitable, since crucial surveillance data can be missed.

An interesting implementation of H.264/AVC for video streaming has been proposed by Antonios Argyriou in [3, 2], where the primary contribution relies on the introduction of a new transport layer protocol called Stream Control Transmission Protocol (SCTP). Due to the message oriented nature of this protocol, it is particularly suitable for handling multiple streams and multi-client access to videos. However, our implementation based on the stack UDP/IP (details are in Section 3.1) provides us with a efficient tool able to meet all the requirements reported in Section 1. In fact, the potential benefits of SCTP are not necessary for our application, where a single video (with only video data and no audio) has to be transmitted to a single receiver, hence not requiring data multiplexing.

Yang *et al.*in [12] addressed the problem of tuning playback frame rate in order to obtain good video quality in live

Tool	PROS	CONS
Windows Media Player for PPC	* Very good for entertainment * Good video quality and fluidity * Decoder complexity sustainable for embedded platforms * free of charge	* no open source * too high latency
Real Media Player for PPC	* same as above * server side is not free	* no open source * high latency
Skype		* video is not supported in mobile version
Quicktime		* mobile not supported
VideoLan for PPC	* open source (based on FFMPEG)	* unofficial nightly-build version for PPC
TCPMP	* open source (based on a portion of FFMPEG)	* network streaming not supported
JM libraries	* well-documented H.264 codec	* no optimized code, too slow
IPP libraries	* best known performance for H.264 decoder	* not freeware/no open source * works only on Intel processors
FFMPEG libraries	* already tested and used by us in [5]	* no ready-to-use solution for PPC

Table 1. Analysis of the state of the art on video streaming tools (last update: May, 2007).

video streaming over networks prone to delays and packet losses; this interesting work can be considered a theoretical work complementary to the architecture proposed in [5], which is going to be extended in this present work.

3. System Architecture

3.1. System Overview

The architecture of our system for live video streaming is basically made of four fundamental parts: the *encoder unit*, which also performs video grabbing; the *video codec technology*, for efficient video compression on low bandwidths; the *network layer*, comprehensive of the communication protocols and of the radio mobile services; the *decoder unit*, responsible also for displaying the video on the screen.

The topic of this present work is an in-depth analysis of the decoder unit specifically designed for PDA processing units. All the other parts, that are just briefly hinted here, are described with full details in [5]. It is important to notice that we have designed the PDA decoder unit in order to be completely compliant with standard H.264 streams.

The encoding is performed in real time, on a standard x86 PC, grabbing live or recorded video data at constant frame rate. All internal data buffering operations have been carefully optimized in order to remove or minimize any unnecessary delay; at the same time, attention was paid to avoid frame skipping as much as possible. The video compression is based on H.264 codec and the reference software is the X.264 open source code.

The adopted communication protocol is UDP, with 1742 bytes long datagrams that encapsulate raw H.264 compressed data; this solution showed excellent results in both latency and reliability. The former is rather foreseeable since UDP datagrams have simpler structure and prioritized dispatch in congested networks compared to TCP segments. The second result might sound surprising: we must deduct that the reliability of the network communication is due to the ARQ mechanism implemented on the Radio Link Control (RLC) of the GPRS. Given these considerations raw UDP was preferred to RTP/UDP, also because the application streams one single video without any audio or text data to be synchronized with.

3.2. Decoder Subsystem: the implementation

Fig. 1 shows a sketch of the PDA decoder unit that will be detailed in the following sections.

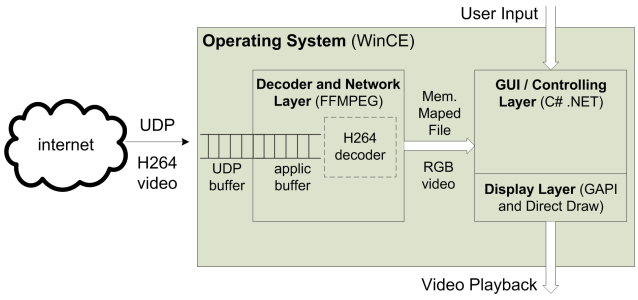


Figure 1. The PDA Decoder Unit

3.2.1 Operating System

The operating system used for our application is Windows CE. Linux for embedded systems has the important advantage of being open source, thus enabling low level control on memory, network and management of processes and services. Such degree of freedom in the tuning of the operating system is desirable since the decoding of compressed video is quite CPU intensive and processing power of PDAs needs to be accurately managed in order to succeed in doing real-time video streaming. Unfortunately, the limited

support of most of the devices and peripherals (such as GSM/GPRS modems) prevented us from adopting this solution until now.

3.2.2 GUI/controlling layer

For ease of implementation, we decided to develop the GUI/controlling layer on C#, making use of the Microsoft Compact Framework .NET. The use of such framework is possible since the tasks of this part are not intensive. As depicted in Fig. 1, this layer is a self-standing application that directly handles the display of the video (see Section 3.2.4) but decoupled from the decoding part.

3.2.3 Decoder and network layer

According to our requirements and to the analysis shown in Table 1, the only feasible, efficient and open-source solution for the decoder is FFMPEG, which is the core of this layer. FFMPEG has been cross-compiled with a suitable modification of the open source GCC compiler.

This layer also handles the video down-streaming from the network. Further details on this specific task are provided in Section 3.3. As abovementioned, the GUI and decoding layers are kept completely detached (i.e., they run on two separated processes), so that each processing flow will not interfere with the other (for example, when the GUI layer calls the garbage collector).

However, there is a need of communication for handing over the video frames from the decoding layer to the GUI/display layer. Since a QQVGA (160x120) video, 24bits RGB colors at 10fps, would generate a 4.6Mbps bandwidth communication, it is evident that the Inter-Process Communication (IPC) must be extremely efficient to avoid frame skipping and preserve video fluidity. Several different solutions to this problem have been tested: data exchange through either a UDP local loop-back or file system has failed since both these methods could not sustain such high data transfer rate without compromising the overall performance of the system. Even worse, using the file system for IPC would end up deteriorating soon the hardware support, based on Flash memories. To overcome to these limitations we used *memory mapped files* (MMFs), i.e. a virtual file on RAM memory. This approach allows to achieve the best performance and also avoids the use of the Flash-based file system.

3.2.4 Display layer

Since the GDI+ functions, available with the managed interface of Microsoft .NET framework, are not meant for video rendering at full screen and at high framerate, our display layer is based on low-level primitives that write video data directly on the graphic card of the device. GAPI (Game API) provided by Microsoft is an excellent solution, but since the maximum supported resolution is QVGA, the support for Direct Draw has been included as well. For a further speed up of this layer, we made use of pre-calculated Look-Up Tables (LUTs) for image rescaling and 90-degrees flipping, used to fit the desired playback frame size and orientation.

3.3. Decoder Subsystem: network buffering and playback control

The correct management of the UDP buffer plays an essential role in order to achieve low latency and good fluidity in the video playback. Since Windows CE does not allow to query the occupancy of the UDP buffer, an application buffer on top of the UDP operating system buffer was added (see Fig. 1). An algorithm for dynamic adaptation of the application buffer size was implemented, in order to keep its level of occupancy controlled and to avoid buffer overflows.

In [5] we proposed an algorithm for tweaking the playback frame rate in order to optimize the latency and the playback fluidity. However, this work could not be implemented on a PDA, given its limited computational power. For this reason the PDA decoder unit employs a light-weighted control based on the key task of keeping the UDP buffer occupancy as low as possible (but always greater than zero) in order to reduce the playback interruptions due to buffer underflows.

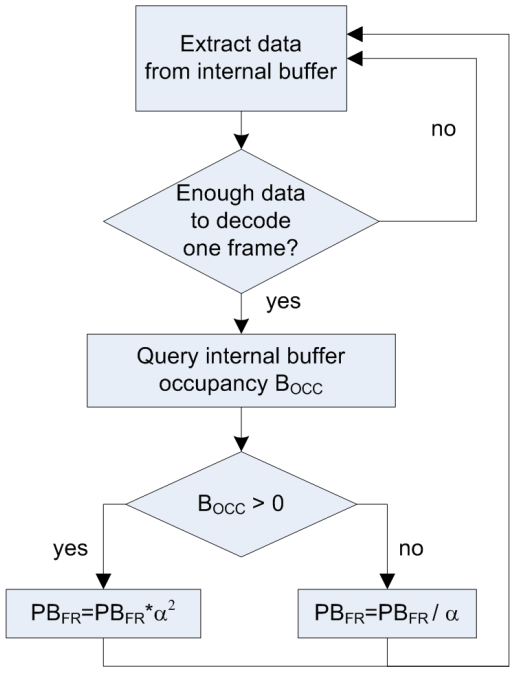


Figure 2. Dynamic Playback Control

The control flowchart is reported in Fig. 2. The α

value is few millesimals above the 1 and allows to modify the playback framerate. If the buffer occupancy B_{OCC} is greater than zero, the control increases the playback framerate PB_{FR} with a value equal to α^2 in order to have a quick reaction, while in the case of occupancy to zero the framerate is divided by α that, being α slightly above one, means a slow reduction of the framerate. Analytical results will be given in Section 4. This control does not claim to be an optimal algorithm for playback frame rate adaptation (for this, refer to [5] and [12]) but our goal is to verify that even on reduced-power processors this simple adaptive control can significantly increase the stability of the video playback.

4. Experimental Results

4.1. Experimental Methodology

Measuring the latency in an analytical way is not a trivial task. An approximate measurement could be obtained by synchronizing the encoding and the decoding unit on the same time server, and modify the encoder so that it dispatches, together with the encoded frames, also the timestamp of their grabbing. Then, the decoding unit detects the time of display of a decompressed frame and deducts the latency by time differencing. This procedure has several drawbacks: the synchronization with the time server must be frequent, in order to have precise time gap measurements; this would be a waste of both CPU cycles and bandwidth; then embedding a timestamp for each frame would result in further bandwidth waste; finally, this kind of measurement requires mofifications to core functions of the video grabbing, networking and displaying, therefore it is only feasible on open source code and cannot be employed on closed systems such as Windows Media and Real Networks systems, that we want to compare with.

Consequently, an alternative way to measure the latency must be used. In our experiments, we used the following methodology: on a recorded video, the frame number was superimposed on each frame. We then played the video both on the encoding unit, and, after having it passed through compression and streaming, also on the PDA.

Let us call $\bar{t}$ a given time instant, $FN_{enc}\left(\bar{t}\right)$ and $FN_{dec}\left(\bar{t}\right)$ the frame number shown on the encoder and decoder respectively. Let us define $\bar{t}^*$ as the time such that $FN_{enc}\left(\bar{t}^*\right) = FN_{dec}\left(\bar{t}\right)$. It holds that $\bar{t}^* < \bar{t}$ and exploiting the definition of latency ($L\left(\bar{t}\right) \equiv \bar{t} - \bar{t}^*$), we can write it as follows:

$$L\left(\bar{t}\right) \equiv \bar{t} - \bar{t}^* = \frac{FN_{enc}\left(\bar{t}\right) - FN_{enc}\left(\bar{t}^*\right)}{GR} = \frac{FN_{enc}\left(\bar{t}\right) - FN_{dec}\left(\bar{t}\right)}{GR} \quad (1)$$

where GR is the video grabbing rate at the encoder side and it is assumed to be constant. In this way, the generic time t can be approximated with $\frac{FN_{enc}(t)}{GR}$.

This procedure needs to embed frame numbers directly on video frames: using plain numbers can be problematic due to the distortion introduced by strong image compression. For this reason, a code-based number representation would be preferable. In our case, we superimposed, on a small portion of each image, the frame number coded in binary. More specifically, 10x10 pixels blocks of white and black colour have been used to code 0s and 1s, respectively. Fig. 3 shows two examples of the used methodology. The leftmost-upper two blocks are used for calibration purposes.

(a) (b)

Figure 3. Examples of the experimental methodology used to measure latency.

We started a video streaming session from PC to PDA and played the video on the screens of both sides; at the same time we filmed both screens with a high-framerate camera. The resulting video is processed with simple image processing algorithms to automatically compute FN_{enc} and FN_{dec} and obtain latency using Eq. 1. This tool is very flexible since it can measure the latency also on closed systems such as Windows Media and Real Networks.

We can compute the number of lost frames $LF\left(\bar{t}\right)$ using the following equations:

$$LF\left(\bar{t}\right) = \sum_{j=0}^{\bar{t}} \varphi(j)$$
$$\varphi(j) = \begin{cases} 0 & \text{if } \Delta FN_{dec} \leq 1 \\ \Delta FN_{dec} - 1 & \text{otherwise} \end{cases} \quad (2)$$

where $\Delta FN_{dec} = FN_{dec}(j) - FN_{dec}\left(j - \Delta t\right)$ and Δt is the discrete sampling period of the acquisition camera. In other words, given that the encoding framerate is constant and that the framerate of the acquisition camera is much higher than the encoding and decoding framerates, if two successive decoded frames contain the same number or successive numbers it means that no frames were lost.

Frame losses can be due to either network datagram losses (this event would most likely produce a drop of several consecutive frames, since with high compression rates and limited frame size, a datagram usually contains several frames) or frame skipping due to compression, or decoder

frame overwriting (frames are decompressed but never get displayed, due to a lack of correct synchronization between the decoder and the displaying processes). In Section 4.3 evidence of frame losses due to compression skipping and to decoder frame overwriting will be given.

In order to measure the video quality at the receiver side, we calculate the PSNR (Peak Signal-to-Noise Ratio) for each received decoded frame compared to its uncompressed version. Moreover, the number of lost frames is taken into account.

4.2. Testing Conditions

We compared our system with Windows Media and Real Networks systems under different testing conditions.

Three different PDAs have been used for the tests (i-Mate JasJar, WM 5.0, CPU: Intel Bulverde, 520Mhz; i-Mate PDA2k, WM 2003, CPU: Intel PXA263, 400Mhz); the results did not show relevant differences by using different PDAs.

The H.264 codec used in our application has been tested in two different profiles: a *baseline* (to achieve low latency at the cost of low quality) and a *high* profile (for best video quality at the cost of higher latency). The high profile contains several enhancements including the use of CABAC encoding, wider reference window for B frames, deeper analysis for macro-block motion estimation, finer subpixel motion estimation and better rate distortion algorithms.

The tests were performed at three different bitrates: 20 kbps, that uses most of the available bandwidth of GPRS; 10 kbps using half of the available bandwidth of GPRS; 5 kbps in order to simulate video streaming over a TETRA (TErrestrial Trunked RAdio) network.

Video sequences for the tests are at 10 fps and QQVGA resolution. They last approximately 120 seconds and contain scenes with different type of motion: reduced (moving persons but fixed camera), medium (moving camera) and extreme (shaking camera).

4.3. Video Latency

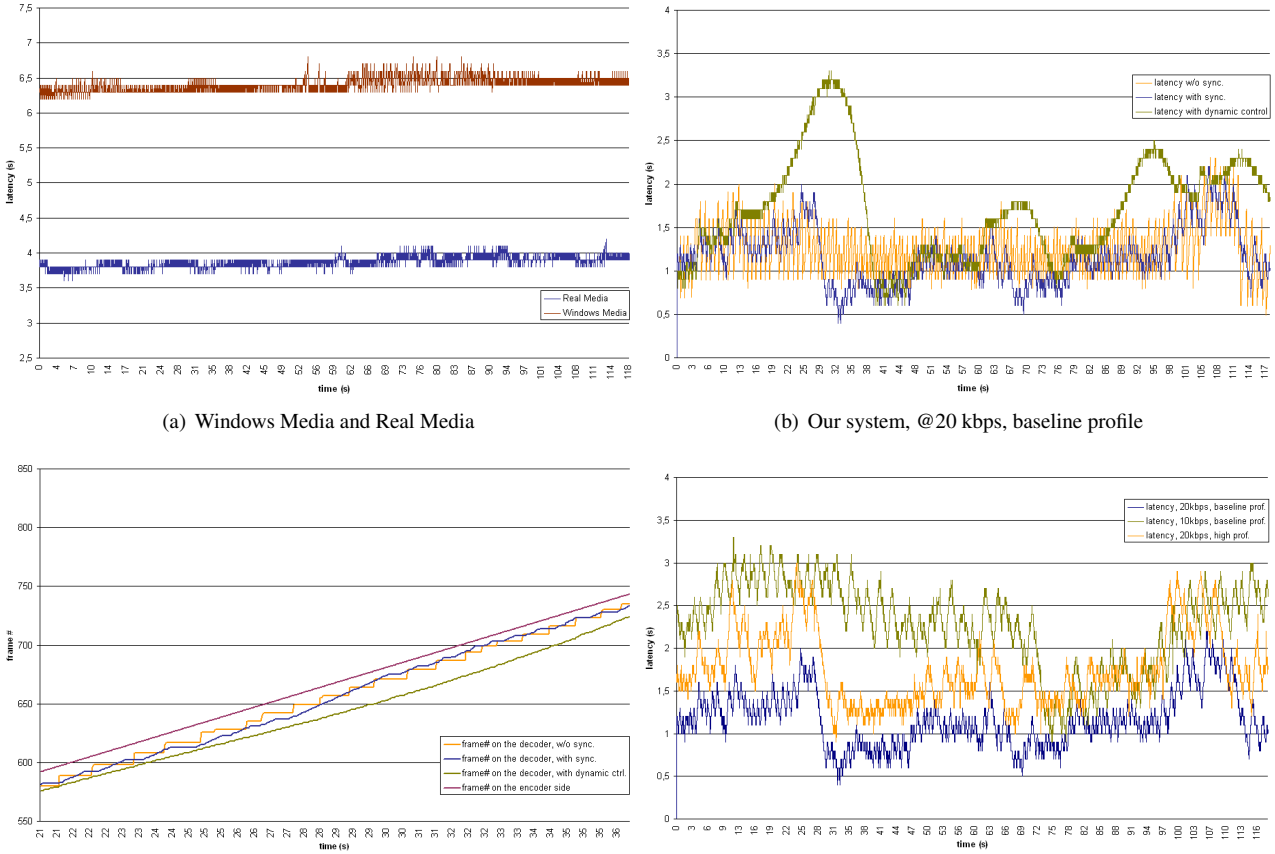
Fig. 4(a) shows the comparison in terms of latency between Windows Media and Real Media, encoding videos at 20 kbps. Both these systems show an almost-constant, rather-high latency. Moreover, Windows Media shows latency scattering from second 60 to second 90, corresponding to the part of the video sequence where the camera is moved very quickly. Moreover, lost frames are concentrated in the part of the video in which the camera is in motion.

Fig. 4(b) plots the latency of our system, encoding video at 20 kbps with the H.264 baseline profile. The orange plot shows the latency when the decoder and the GUI/display layers are running asynchronously. This generates a massive presence (79%) of frame losses, due to overwriting. The plot shows sharp and regular peaks due to a twofold reason: buffer underflows (sharp latency raises), and very fast playback (sharp latency falls). Introducing a simple syncronization on the decoder that avoids the frame overwriting through waiting up to a maximum of 50 ms, the frame losses are reduced to 2%: this is the blue plot, that shows smaller peaks as expected. It is worth noting that the introduction of a syncronization between the display and the decoder increases the dependency of the latency on video sequence complexity: for example, the latency suddenly drops around second 28, when the camera starts to move. This can be explained because of the tolerance of the bitrate control: the more complex the scene is, the higher the encoding bitrate and the lower the time to fill up and deliver a datagram. The opposite effect is visible around second 96, when the camera stops moving. In addition, Table 2 shows that the introduction of the synchronization has a positive effect on the average latency. The green plot of figure 4(b) shows the latency when the dynamic control is turned on ($\alpha = 1.002$): since the buffer is initially empty, it reduces the playback framerate of a α factor each frame; after a few seconds (approximately 15), the playback framerate has reached a critical value and the latency starts to increase until the buffer occupancy becomes definitely greater than zero: the dynamic control then reacts quickly, increasing the playback framerate of α^2 each frame, thus reducing latency. It's worth noting that the dynamic control drastically reduces the presence of peaks in the latency.

	Avg	**Std.Dev.**
Windows Media @20 kbps	6.42	0.14
Real Media @20 kbps	3.87	0.10
Ours @20 kbps, baseline, w/o sync	1.25	0.28
Ours @20 kbps, baseline, with sync	1.16	0.30
Ours @20 kbps, baseline, dyn. ctrl.	1.76	0.39
Ours @20 kbps, high prof., with sync	1.68	0.58
Ours @10 kbps, baseline, with sync	2.29	0.47
Ours @20 kbps, bas., sync, mob-to-mo	2.96	0.49

Table 2. Latency measurements: averages and standard deviations

Fig. 4(c) shows the frame number trend of the streams of Fig. 4(b) in a short time interval. The plot clearly shows the smoothness in the playback introduced by the dynamic control. Eventually, Fig. 4(d) shows the latency measured for our system, with synchronization between decoder and display but without dynamic control, in the following conditions: baseline profile at 20 kbps, high profile at 20 kbps and baseline profile at 10 kbps. As expected, the high profile increases the latency. Also the 10 kbps stream latency



(a) Windows Media and Real Media

(b) Our system, @20 kbps, baseline profile

(c) Frame number trends in our system

(d) Our system, with synchronization decoder/display

Figure 4. Comparison in terms of latency. This figure is best viewed in the electronic paper version.

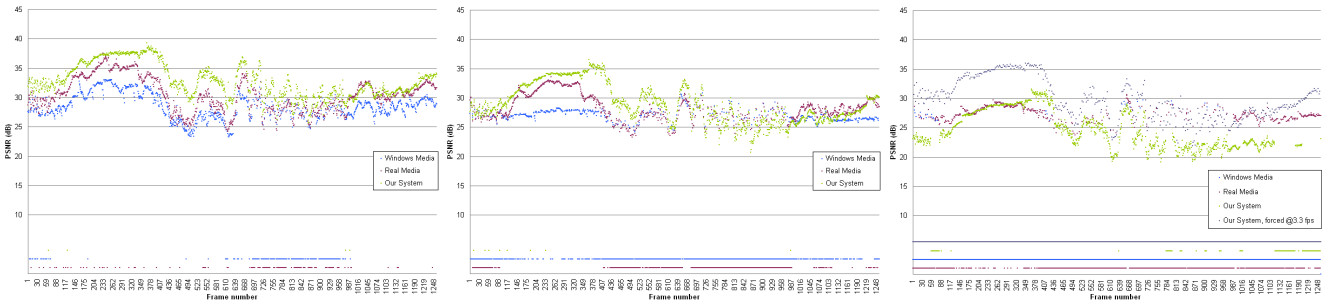
is higher: this is because the reduction of the encoding bitrate increased the time to fill the UDP datagram (that did not change in size). We also measured latency in a mobile-to-mobile (specifically laptop to PDA) setup (see Table 2): the plot of the latency is similar to the other cases, but the average and the standard deviation tend to increase. This setup introduces a further degree of instability in the network communication, due to the additional step of the video data flow on radio mobile channels.

4.4. Quality of video frames

To compare the quality of video frames, we calculated the PSNR of the compressed frames on the three systems. Our system was configured in high profile (consider that even with high profile the latency is much lower than Windows Media and Real Media). Fig. 5 shows the results at 20 kbps, 10 kbps and 5 kbps, with superimposed dots (at the bottom of the graphs) representing the lost frames. It is important to notice that the stronger the compression becomes, the higher the frame loss rate is. It is evident that our system outperforms, on average, the other two, especially in terms of lost frames. Table 3 reports a summary of the percentage of lost frames and the average PSNR. Our system could sustain 10 fps even at 5 kbps, but, as expectable, the fluidity is maintained only at the cost of PSNR. Forcing our encoder to skip 2 frames on 3, PSNR increases significantly (Fig. 5(c)): the percentage of lost frames is 67.48% (consider that 66.6% was due to the forced frame skipping at the encoder side), but the average PSNR is 29.76 dB.

5. Conclusions

This paper reports the extension to the previous work reported in [5] to include the decoder of H.264 live streams on a PDA. The analysis of the state of the art has shown the limitations of both commercial and scientific tools for live video streaming on PDAs, with the strict requirements on low latency, good video fluidity and the best image quality possible. Therefore, a complete, open-source solution based on X.264 and tailored to work on low-capacity networks with low latency has been developed and tested. The extensive comparative analysis demonstrates that our ap-



(a) 20 kbps (b) 10 kbps (c) 5 kbps

Figure 5. Comparison of video quality in terms of PSNR. The dots in the lower part of each graph indicate frame losses. This figure is best viewed in the electronic paper version.

	PSNR (dB)			% of lost frames		
	20 kbps	10 kbps	5 kbps	20 kbps	10 kbps	5 kbps
Windows Media	28.52 (2.14)	27.02 (0.97)	27.81 (1.01)	17.27%	56.73%	96.11%
Real Media	30.39 (2.80)	28.59 (2.24)	27.36 (1.78)	8.93%	30.04%	60.01%
Our System	32.80 (2.77)	28.93 (3.28)	24.47 (3.01)	0.32%	0.64%	15.86%
Our System, forced @3.3 fps	n/a	n/a	29.76 (3.28)	n/a	n/a	67.48%

Table 3. PSNR and percentage of lost frames. PSNR is expressed on average (and standard deviation in round brackets).

proach outperforms existing commercial tools in both latency, percentage of lost frames and average PSNR.

Acknowledgments

This work is partially supported by the project FREE SURF (FREE SUrveillance in a pRivacy respectFul way), funded by MIUR (project nr. 2006099482), and European VI FP, Network of Excellence DELOS (2004-08) on digital libraries, sub-project Multimedia Interfaces for Mobile Applications (MIMA).

References

[1] Advanced video coding for generic audiovisual services. Technical report, ITU Rec. H624/ISO IEC 14996-10 AVC, 2003.

[2] A. Argyriou. A Novel End-to-End Architecture for H.264 Video Streaming over the Internet. *Telecommunication Systems*, 28(2):133–150, 2005.

[3] A. Argyriou and V. Madisetti. Streaming H.264/AVC video over the Internet. In *Proc. of 1st IEEE Consumer Communications and Networking Conference*, pages 169–174, 2004.

[4] G. Conklin, G. Greenbaum, K. Lillevold, A. Lippman, and Y. Reznik. Video coding for streaming media delivery on the Internet. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3):269–281, Mar. 2001.

[5] G. Gualdi, R. Cucchiara, and A. Prati. Low-latency live video streaming over low-capacity network. In *Proc. of IEEE Int'l Symposium on Multimedia*, pages 449–456, 2006.

[6] M. Guo, M. Ammar, and E. Zegura. V3: a vehicle-to-vehicle live video streaming architecture. In *Proc. of IEEE Intl Conf on Pervasive Computing and Communications*, pages 171–180, 2005.

[7] K. Lim, D. Wu, S. Wu, R. Susanto, X. Lin, L. Jiang, R. Yu, F. Pan, Z. Li, S. Yao, G. Feng, and C. Ko. Video streaming on embedded devices through GPRS network. In *Proc. of IEEE Intl Conference on Multimedia and Expo*, volume 2, pages 169–172, 2003.

[8] Z. Liu and G. He. An embedded adaptive live video transmission system over GPRS/CDMA network. In *Proc. of Intl Conf. on Embedded Software and Systems*, 2005.

[9] J. Lu. Signal processing for internet video streaming - a review. In *Proc. of Conf on Image and video communications and processing*, pages 246–259, 2000.

[10] M.-T. Lu, C.-K. Lin, J. Yao, and H. Chen. Complexity-aware live streaming system. In *Proc. of IEEE Int'l Conference on Image Processing*, volume 1, pages 193–196, 2005.

[11] A. Puri, X. Chen, and A. Luthra. Video coding using the H.264/MPEG-4 AVC compression standard. *Signal Processing: Image Communication*, 19:793–849, 2004.

[12] Y. Yang, M. Lu, and H. Chen. Smooth playout control for video streaming over error-prone channels. In *Proc. of IEEE Int'l Symposium on Multimedia*, pages 415–418, 2006.