

DDXVIDEO: A Lightweight Video Framework for Autonomous Robotic Platforms

Elliot Duff

CSIRO ICT Centre

PO Box 883, Kenmore, Qld 4069, AUSTRALIA

Email: firstname.lastname@csiro.au

Abstract

This paper describes the implementation of a lightweight video framework for autonomous robotics platforms. It is based upon DDX (Dynamic Data eXchange) which is our third generation real-time publish-subscribe software (event-based middleware). Computational resources on autonomous robotic platforms can be very limited, and thus it is essential that mechanisms for access to video data have as little impact upon the computation load as possible.

1 Introduction

We are particularly interested in the navigation of autonomous outdoor robotic platforms: including submarines, helicopters and ground vehicles. With advances in computer hardware over the last few years, we have become increasingly interested in systems that are capable of real-time video processing. As with most research organizations, we have a number of researchers who are performing different tasks on the same platform: i.e. optical flow, stereo, scene segmentation, beacon localization etc. To solve the task of navigation, many of these tasks need to be solved in parallel. Therefore it is important that we establish a common framework for video acquisition and processing. Here is a list of desirable features for such a framework:

1. ability to decouple video acquisition from video processing and display;
2. support different video sources (analog or digital camera, pre-recorded or processed video);
3. ability to synchronize video with other sensors and control data;
4. ability to record video without compression or dither;
5. stream video to remote locations over wireless link; and
6. be simple and transparent with very low overheads.

The traditional method has been to develop a monolithic process, with separate threads that adhere to a strict API and are

given appropriate resources. This is the approach taken by the Player/Stage [Vaughan *et al.*, 2003] platform with plug-gable software modules. Although this has a number of advantages (i.e. code reuse) we would prefer a more decentralized approach. There are a number of alternative robotic software platforms that could be used: YARP [Metta *et al.*,] and ORCA [Brooks *et al.*,] using TAO CORBA [Schmidt,] or CARMEN [Montemerlo *et al.*,] using IPC [Simmons and James,].

The TAO CORBA A/V [Munee *et al.*, 1999] streaming service is based upon a distributed system which has been modified to provide local support. For several reasons discussed in [Corke *et al.*, 2004], we would prefer the reverse: a system that is based upon local support with distributed features added on. Given this preference and the overall complexity of CORBA, we decided to investigate the possibility of modifying our own software to support video.

The remainder of this paper is structured as follows. Section 2 describes the implementation of supporting video in DDX and the modifications that were made to increase its efficiency. Section 3 describes some of the experimental results, and Section 4 concludes and presents future directions.

2 Implementation

For over 10 years we have been building complex robotic systems. Over time we have developed a series of software platforms (middleware) to facilitate this endeavor. Our third generation middleware software is called DDX (Dynamic Data eXchange). This is event-based middleware that provides distributed real-time publish/subscribe access to sensor and control data. It does this through an efficient shared memory mechanism managed by a **store**. The **store** provides a naming service to data in shared memory. Stores on multiple machines can be linked by means of a global **catalog** and data is multicast between the stores when the data is requested (subscribed to). DDX also has a number of service applications, including the ability to log and replay data in real-time. The implementation of DDXVIDEO consists of an agreed format in the **store** and a suite of applications (see Figure 1).

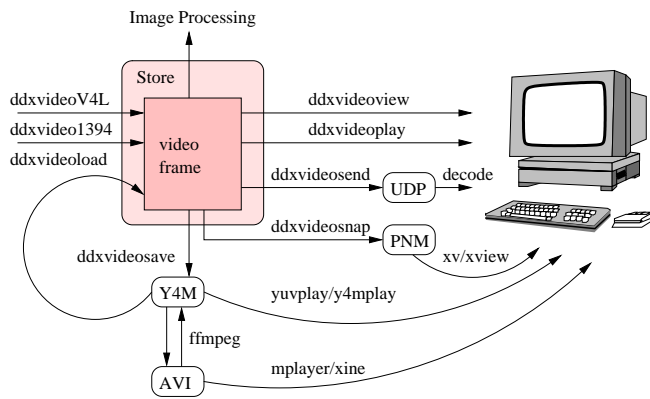


Figure 1: DDXVIDEO applications.

2.1 Video Format

Typical video formats can either be packed or planar, use RGB or YUV colourspace, and have different levels of sub-sampling. Analogue frame grabbers (such as the BT878 chipset) processed all incoming video as YUV422. In this packed colourspace the U and V components (chroma) are down-sampled 2 to 1 in the horizontal direction. This means that if an RGB colourspace is requested, the image will only contain colour information that existed in the original YUV422 image. Another video format is YUV420. In this colourspace, the U and V (chroma) are downsampled 2:1 in both the vertical and horizontal direction. This differs from YUV411, which is downsampled 4:1 in the horizontal direction. Whilst the YUV420 format has half the chroma resolution of the YUV422 format, it is the format preferred by JPEG and MPEG compression routines and by XVideo to display video under X11. This is an important factor when the speed of encoding and decoding will have an impact upon the computational load. Perhaps the most important factor was the requirement to be able to record and playback the video stream without compression. The most popular uncompressed video format is YUV4MPEG [<http://mjpeg.sourceforge.net>], Y4M for short. Whilst not perfect, the YUV420P format is a reasonable compromise between resolution, compatibility and speed, and thus was chosen as the default format for video in the **store**.

The shared memory data structure is shown in Figure 2. There are three data structures in the **store**. The first is DDX_VIDEO, which contains a header and three video planes (Y,U,V). The Y is the luminance, whilst the U and V make up the chrominance at half the resolution of the Y plane. To accommodate multiple video streams that need to be synchronized (such as stereo) each video stream is placed on top of one another in memory. This is represented by the number of fields in each frame. This is shown in Figure 2 where there are three video fields per video frame.

The data structure is defined in DDX by the following declaration:

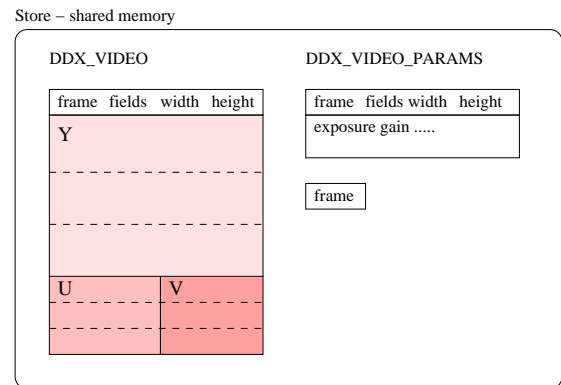


Figure 2: Video data format in store.

```
DDX_STORE_TYPE( DDX_VIDEO,
struct {
    int frame; // Frame sequence number
    int fields; // Number of fields in frame
    int width;
    int height;
    char y[MAX];
    char u[MAX/4];
    char v[MAX/4];
} );
```

The second data structure contains camera parameters which are updated whenever the camera settings are modified. Typically, the camera is controlled by a simple RPC request to the corresponding application. The third data structure is a simple integer frame counter. This is used to synchronize normal DDX data with video data.

2.2 Access to Video data

Access to the video data is provided by standard DDX calls to the **store**:

```
DDX_STORE_ID * storeId = NULL;
DDX_STORE_ITEM * itemPtr = NULL;
storeId = ddx_store_open(NULL, port, timeout);
itemPtr = ddx_store_lookup_item(storeId, "video", NULL, 0);
```

where **ddx_store_open** is used to make a connection to the local **store**, and **ddx_store_lookup_item** is used to return a handle to a named item. In this case to *video*. The item handler is then used to read the video data.

```
DDX_VIDEO video;
video = ddx_store_read(itemPtr, &video, &ts, 1.0, 1);
```

This function can be used in blocking or non-blocking mode. In either case, a local copy is made of the current video frame. To guarantee the integrity of data, the store places a mutex around the read and writing of data to shared memory. This prevents one application writing to memory whilst another is reading. This behaviour is critical for most sensor and control data, however if we only wish to “spy” on the current data, then this locking can be excessive. To prevent unnecessary locking and copying of data, functions to DDX were introduced that provide direct access to the store data. This is done with the following commands:

```
DDX_VIDEO * video;
video = ddx_store_var_pointer(itemPtr);
ddx_store_read_direct(itemPtr, &ts, 10.0, 1);
```

where, **ddx_store_var_pointer** provides a pointer to the actual video shared memory. The **ddx_store_read_direct** acts as a semaphore informing the application that someone has just finished writing to the store. The direct access to video memory is used by applications (such as display applications) that will not be significantly affected if the video memory were altered whilst displaying the image. This type of behavior can reduce the computational load. Of course, the fact that we now have direct access to the current video frame is not without risk, and it is up to the application writer to exercise caution.

2.3 Video Applications

There are a number of application that have been written that can load video into the **store**:

ddxvideoV4L use V4L (video4linux) drivers to acquire images from analog framegrabber.

ddxvideo1394 use dc1394 (firewire) drivers to acquire images from firewire bus.

ddxvideoSVS use SVS libraries [] to acquire range images from SVS camera (stereo).

ddxvideoload read Y4M (yuv4mjpeg) file and load into store at specified time-steps.

And a number of applications that display or save video from the **store**:

ddxvideoplay display video stream locally

ddxvideoview display video image remotely (step though each frame)

ddxvideosave write videostream as Y4M file (yuv4mjpeg)

ddxvideosnap write single frame as PPM file

ddxvideosend use libavcodec to encode image and send out through UDP port.

decode to decode UDP packets and display on remote screen.

Here is an example that will acquire video at 5Hz in CIF resolution.

```
catalog &
store -m 8M &
ddxlog -c logger.conf &
ddxvideoV4L -camera 2 -skip 5 -size CIF -name video &
ddxvideoview -name video &
ddxvideosave -name video -output /tmp/test.y4m &
```

Here the **catalog** and **store** are started with sufficient shared memory to handle video. Video is then acquired on a specified channel, with a specified rate and size, and copied into the store with a specified name. In this

case it is called *video*. The current *video* frame can be displayed with **ddxvideoview**. In theory, one could use **ddxlog** to record the video stream. However, this log file is only compatible with DDX. The preferred option is to use **ddxvideosave** to record a Y4M file. Once recorded the Y4M file can displayed with **yuvplay** or **mplayer**. It can also be transcoded (converted to a number of video formats) with **ffmpeg** or **mencoder**.

```
mencoder -o test.avi -ovc lavc /tmp/test.y4m
```

One of the features of the Y4M file is that it is possible to record the time step between each video frame. This time-step is used to replay the video back to the store in real-time, which can be used to test the performance of the image processing routines.

```
ddxvideoload /tmp/test.y4m
```

Furthermore, if the frame counter is recorded in a log file along with named sensor data, it is possible to synchronize log data with video data.

2.4 Video Streaming

Although the store can be configured to handle large data structures (such as images), the sharing of data between stores is currently restricted to the size of a single UDP packet (8k). Therefore to send the video sequence to another computer it is appropriate to bypass the store and broadcast the video directly. Unfortunately an uncompressed PAL video stream consumes approximately 256Mbps, which is far beyond the capabilities of the WiFi network that we have between our autonomus platforms. Although there are many applications that can compress and stream video (FFmpeg, Mbone, Mash, VideoLan, Darwin, RealSystems) they are all designed to synchronize the video and audio streams.

To achieve synchronization it is important that the packets arrive in the correct order and at the correct time. Since neither of these conditions can be guaranteed on the Ethernet, buffering is required at the receiving end to reorder and request that lost packets be resent. This buffering can have a significant effect upon latency, especially in wireless environments which can have a low bandwidth and high packets loss. Since is our intention to use the remote video stream to control the robot it is critical that the latency be as small as possible (less than 100ms). Fortunately, since we are not interested in the audio stream, it possible to write our own streaming software without buffering.

```
ddxvideosend -port 8000 &
```

In this example, the video stream can be decoded and viewed on another computer with an application called **decode**. Since, by default, the video is multicast on a UDP socket, any number of users on site can “tune-in” into port 8000 to watch the streaming video. This is a very convenient way to keep track of what is happening on our autonomous platforms without imposing additional load on the system. Thus we need to choose a video CODEC that provides reasonable



Figure 3: Omnidirectional Megapixel Image.

quality and robustness, with low latency and bandwidth, without consuming too many computational resources. Motion JPEG is one of the first streaming technologies to be used. It is robust because there is no interframe dependence, ie. frames can be lost without significant impact, however significant compression is required to get below the 8K threshold of UDP transmission. Higher compression can be achieved with interframe compression - where there is compression from one frame to another (ie. MPEG 1,2 & 4). The most appropriate for streaming is MPEG4 because it have been designed to withstand significant packet loss. Since MPEG4 is robust to packet loss, no attempt is made to recover dropped frames; reorder frames that are out of sequence, or even to impose a time-step between frames. As each frame is received it is decompressed and displayed. In this way latency is kept to an absolute minimum. This applications also has the ability to record the received video sequence to an AVI file.

3 Experimental Platforms

3.1 Ground Vehicle

The ground-vehicle platform (a small ride-on lawn mower) is fitted with five digital cameras: two fisheye, two stereo, and an omni-directional mega-pixel camera. These cameras are connected via Firewire to a single Pentium-M miniITX computer. The output from the omni-directional cameras is shown in Figure 3. Colour segmentation of the UV plane is used to track artificial beacons and segment the road.

3.2 Underwater Platform

The underwater platform (called Starbug) is fitted with a two stereo pairs of analog cameras. The stereo cameras are line

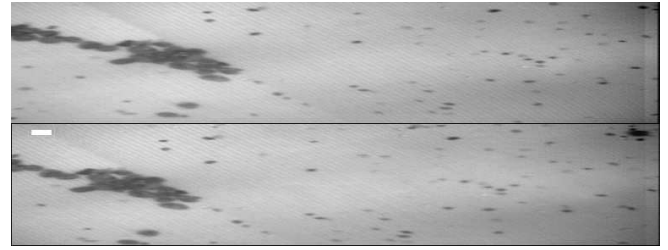


Figure 4: Analog stereo image taken underwater.

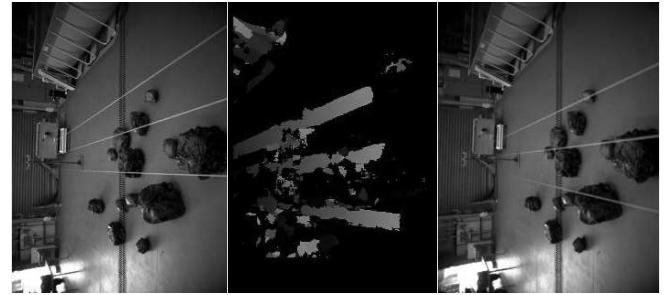


Figure 5: Tertiary image taken from AVS pod.

interlaced and fed into a standard BT878 frame grabber on a 800MHz PC104 Crusoe processor. In `ddxvideoV4L` the video is de-interlaced into the respective fields and placed on top of one another. This is shown in Figure 4, where a number of rocks on the bottom of a swimming pool can be seen. The disparity between the images is used to estimate height. Simultaneously, another pair of forward looking cameras are used for collision avoidance.

3.3 Aerial Platform

The AVS (Air vehicle simulator) is fitted with interchangeable Firewire cameras, connected to a self-powered wireless Pentium-M miniITX. The results from the SVS stereo camera are shown in Figure 5, where the range image is inserted between the left and right image. Another application uses the range image to estimate location of the ground plane.

3.4 Performance

One key requirement of DDXVIDEO is the ability to stream video quickly and robustly over a wireless network. The results of MPEG4 compression at 200kbps over a IEEE 802.11b network is shown in Figure 6, where a packet loss of 30% has been simulated by deliberately dropping every third packet. In this figure, a hand is waived in front of the camera to demonstrate MPEG4's ability to rebuild the image. Note the black squares around the edges of the hand. Similar robustness was exhibited after re-ordering the packets, mixing packets from other streams, and dropping GOP frames. On the AVS platform, the compression software only consumed 5% of the CPU.

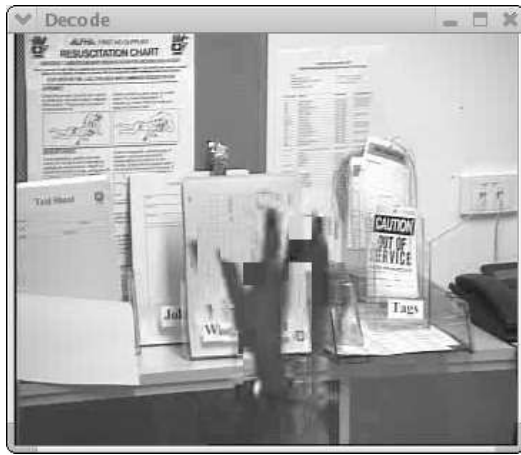


Figure 6: MPEG4 at 200kbps with dropped packets.

Another requirement of DDXVIDEO is the ability to record uncompressed video at selected frame rates. On Starbug a frame rate of 5Hz was required to test optical flow algorithms. The performance of a 800MHz Crusoe with a laptop hard drive was rather disappointing with significant delays (up to 0.8s) every minute or so, but this was found to be related to the EXT3 file system, and was fixed by switching off the journaling function. This highlights the importance customizing software with the operating system and hardware.

4 Conclusion

DDXVIDEO was originally developed to assist with the development of image processing routines on autonomous platforms. Since it is now being routinely used on all of our autonomous platforms (air, land and water) it would be reasonable to conclude that its implementation has been a success. Its success lies in the fact that:

- It provides multiple access to the same video stream with insignificant overhead (just pointers).
- It can store multiple “fields” in the same video frame with guaranteed synchronization.
- It can save video stream in format that is compatible with popular video players
- It is possible to replay video stream in real time, with exactly the same data.
- It uses MPEG4 compression which is efficient and requires minimal bandwidth.
- It uses UDP multicast which has low overheads and very low latency.

Future directions:

- Support for alternative colourspace, with different sizes, compatible with Intel’s IPP.
- Support for off-the-shelf streaming software such as VideoLan’s VLC.

It is interesting to note that other robotic platforms have adopted new transport layers based upon shared memory and UDP: e.g. CRUD in the ORCA platform, and Gazebo in the Player/Stage.

Acknowledgment

The author would like to thank the rest of the CSIRO Autonomous Systems Laboratory who have contributed ideas and have thoroughly tested this software. In particular, Pavan Sikka for his help with DDX and Cedric Pradalier for his help with the 1394 code.

References

- [Brooks *et al.*,] Alex Brooks, Tobias Kaupp, Alex Makarenko, Anders Oreback, and Stefan Williams. Towards Component-Based Robotics. <http://orca-robotics.sourceforge.net>.
- [Corke *et al.*, 2004] Peter Corke, Pavan Sikka, Jonathan Roberts, and Elliot Duff. DDX: A Distributed Software Architecture for Robotics Systems. In *Proceedings of the Australian Conference on Robotics and Automation*, Canberra, Australia, 2004.
- [Metta *et al.*,] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: Yet Another Robot Platform. <http://yarp0.sourceforge.net>.
- [Montemerlo *et al.*,] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. CARMEN Carnegie-Mellon Robot Navigation Toolkit. <http://www.cs.cmu.edu/~carmen>.
- [Munee *et al.*, 1999] Sumedh Munee, Nagarajan Suren-dran, and Douglas Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Proceedings of the HICSS-32 International Conference on System Sciences, Multimedia DBMS and the WWW*, Hawaii, USA, 1999.
- [Schmidt,] Douglas Schmidt. Real-time CORBA with TAO. <http://www.cs.wustl.edu/~schmidt/TAO.html>.
- [Simmons and James,] Reid Simmons and Dale James. *Inter-Process Communication*. Carnegie-Mellon University. <http://www.cs.cmu.edu/~ipc>.
- [Vaughan *et al.*, 2003] R. T. Vaughan, B P Gerkey, and A Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2421–2427, Las Vegas, USA, 2003.