Introduction to Veter project

Andrey Nechypurenko
Munich, Germany
andreynech@gmail.com

Maksym Parkachov
Munich, Germany
lazy.gopher@gmail.com

August 13, 2010

Contents

1 Introduction

About three years ago, as a hobby project, we start developing small vehicle equipped with on-board computer connected to WLan adapter and web-camera. The idea was to make it possible to control the car over Internet by streaming the video from on-board camera to the driver and send control commands back to the car. Figure 1 shows the first version of the car.

Figure 1: Monster Track version 1.0.

As the first version of the car was ready we got an idea to organize races where several drivers could buy a ticket for the race (actually download SSL certificate to connect to the car) and drive against each other. Collected money could be used to pay the prize to winners and make the profit for organizers. We are currently in process of organizing such races and looking for business partners. Please feel free to contact us if you are interested.

While developing the first version of the car, we collect a lot of experience with hardware and software design. Also, there are new embedded platforms appeared on the market which are more suitable for our purposes. That is why, we decide to set new goals for our project which might be more attractive and let us continue improving our hardware and software skills. In particular, we decide to get up in the air and build quad-copter which could be also controlled over the Internet (just imaging Star Wars like races in the 3D maze :-)).

We are reusing a lot of building blocks (software and hardware) from our first car, but there are also some fundamental changes. For example, in contrast to our first car, which was built using Intel Atom based mini-ITX board we decide to use BeagleBoard as on-board computer. The big change at the software side is the move from simple Xvid based video encoding to Gstreamer library. As a first iteration to achieve our new goal, we decide to rebuild the car with the new software and hardware to collect experience with these new platforms. After that we can apply these technologies to build quad-copter.

This document describes hardware and software related aspects of the new car design and organized as following. Section 2 provides the overview of our current hardware setup. Section 3.1 describes the architecture of the vehicle on board application and section 3.2 describes the architecture of the cockpit application used by the driver to control the vehicle.

2 Hardware design

The Figure 2 represents the main hardware components we are using.

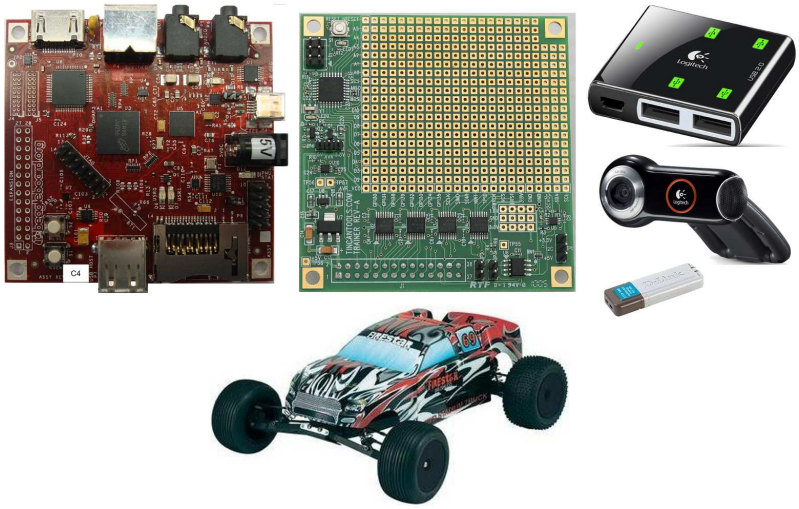

Figure 2: Main parts.

BeagleBoard fits our needs very well because of: a) small form factor; b) low power consumption which in turn reduce size and weight of batteries; c) powerful enough to compress video with h264 codec in real-time using DSP; d) provides enough IO channels such as I2C and GPIO to control attached hardware and e) provides USB ports to connect camera and WLan adapter.

Trainer Board is expansion board designed specially for beagle board and provide the set of useful hardware such as for example voltage level shifters to convert BeagleBoard's 1.8V based IO pins to more widely used 5V level. In addition, there is on-board Atmel AVR micro-controller which can greatly simplify the implementation of the real-time motor control tasks such as PWM signal generation. Despite the convenience of the Trainer Board, we are considering it as a temporary solution for the car. In the future, we are going to use motor controllers with I2C interface directly connected to the BeagleBoard and as a result eliminating the need for additional board. Moreover, we are currently investigating the real-time performance of the Linux kernel with preempt_rt patches on the BeagleBoard to control motors directly from the BeagleBoard without additional micro-controllers. It might reduce the overall system weight and hardware complexity.

Logitech USB HUB is required because there is only one USB host port available on the beagle board. However, it might be possible to force the USB OTG port to work in a host mode with a little bit of soldering. We are currently investigating this option. In case of success, the USB HUB would not be necessary which will greatly reduce system size and weight.

Logitech 9000 Pro USB camera is high quality USB camera. The main reason why we decide for this camera was the bright view angle which is absolutely required to drive the car remotely. In the first version of the car, we start

with Philips ToU Camera. However, after test drives we found that it is very hard to drive the car with such view angle and move to the brighter angle analog camera connected through the USB capturing module. 9000Pro provides almost the same view angle as our analog camera and that is why we decide to use it for the new car design.

D-Link USB WLan adapter is standard 54 Mbit/s adapter with Ralink chipset which is well supported by Linux kernel.

Reely buggy is a little bit smaller then our first car and has better motor (brushless). Size reduction achieved by the move from mini-ITX to BeagleBoard makes it possible to pack all the electronic within this smaller car.

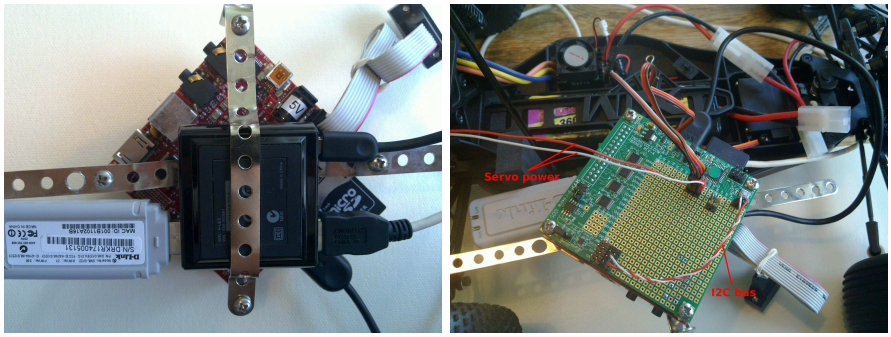

Figure 3: Main parts assembled together

Figure 3 illustrates how these parts are looking assembled together. On the picture at the right hand side there are two connectors for the servo-controllers (steering and acceleration) where signal wire is connected to the AVR digital outputs available on the Trainer Board. In addition, we connect the AVR to the BeagleBoard's I2C bus to send control commands from BeagleBoard to the AVR which acts as a slave on the bus. As was mentioned above, we are planing to use I2C enabled motor controllers for the quad-copter and that is why we want to use I2C to make as less changes as possible in the on-board software when we will switch to these motor controllers.

In addition, there is a separate power supplied for the servos. Initially, we were using the 5V power supplied by the trainer board which is in turn supplied by the BeagleBoard. It does not work because of the strong noise generated by servos. Video quality was very affected by this noise and sometimes even the whole USB subsystem. That is why, separate power for servos is necessary.

3 Software architecture description

The whole system software has two main tasks: a) deliver video stream and sensor data from the vehicle to the remote driver and b) deliver control commands from the driver to the vehicle software and drive vehicle actuators. We assume that the delivery happens over the Internet where it is typical to have two firewalls (and/or NATs) on the client and server side. The Figure 4 represents this setting.

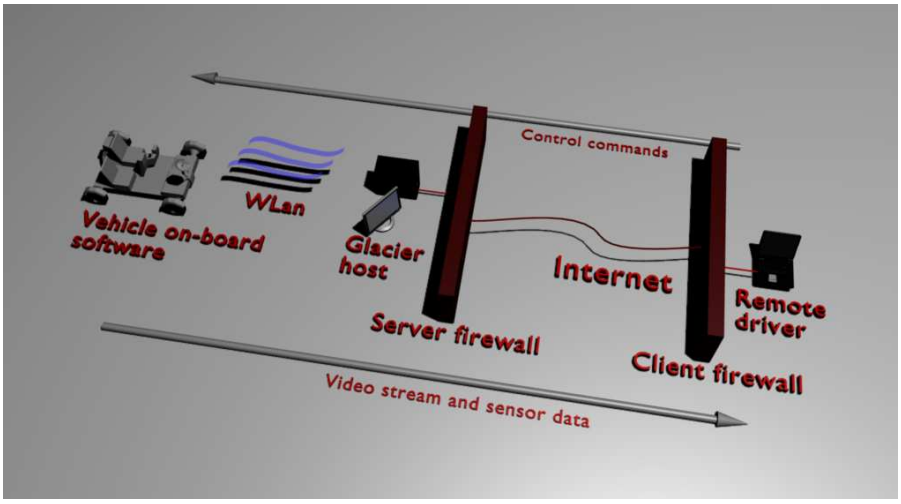

Figure 4: High level system overview.

To provide this functionality two main modules are required: a) driver application which we are calling cockpit and b) vehicle on-board application which we will be simply calling vehicle. In addition, to perform firewall traversal in the secure and efficient way, additional application is required on the server side. To address all communication aspects we are using ZeroC Ice middleware for all our communication needs. The main reasons for this choice are the following:

- Dramatically reduce complexity when implementing complex biderection communication.
- Transparently handles cross-platform issues such as endianess when communicating between vehicle application on ARM (BeagleBoard) and cockpit application running on Intel x86 platform.
- There are two versions of Ice. IceE (E stands for embedded) which has reduced footprint and easier to cross-compile which makes it a great choice for embedded vehicle application. The complete version provides the full set of functionality which we are using for cockpit application. These versions are compatible with each other on communication protocol level.
- Very easy to change the communication protocol. For example, UDP is preferred choice for video transmission. However, it is much harder to solve firewall and NAT related issues with UDP. That is why, in such cases TCP might be preferred. If high security level is required, then SSL encryption might be necessary. Changing between UDP, TCP or SSL is the matter of change endpoint description in the configuration file.
- Ice provides special service application called Glacier which solves firewall/NAT related problems. In this document we will not describe how Glacier works. Ice documentation could be consulted for more information.

- Great scalability in case of many network clients.

The following sections provides more details on vehicle and cockpit applications architecture.

3.1 Vehicle on-board application

The vehicle on-board application (we will call it vehicle in the following text) has the following responsibilities:

- Receive control commands (such as steering and acceleration) from remote driver
- control connected accelerators based on the received commands. In particular, send motor control commands over I2C interface.
- Capture video from camera and compress it in real-time.
- Send captured video to the cockpit application.
- Collect statistic about network bandwidth to perform adaptation in case of changed network conditions. In particular, the frame size could be reduced, compression rate could be increased or even frame rate could be reduced if the bandwidth is not enough to deliver the video on time.

3.1.1 Networking with Ice

We are using ZeroC Ice for communication purposes. The following fragment represents the most important part from the interface definition file.

```
interface StreamReceiver {
  ["ami"] void nextChunk(ByteSeq chunk);
  idempotent QoSReport getQoSReport();
};

interface MotorControl {
  idempotent void setDuties(MotorDutySeq duties);
};

interface RemoteVehicle {
  . . .
  idempotent MotorControl* motorControlInterface();
  idempotent void addStreamReceiver(StreamReceiver *callback);
};
```

Vehicle application implements two of these interfaces:

- *RemoteVehicle* is the entry point to the vehicle functionality and can be used to request the MotorControl interface as well as setting remote callback interface to receive compressed video stream.

- *MotorControl* interface allows remote access to the available motors. Remote application can set the motor duty in percents with just one remote invocation passing the sequence of required motor duties as a parameter. Vehicle application implements this interface. When required duties are received, the percentage is converted into actual motor control commands and sent over I2C to the motor controller. Currently, we are using Trainer Board connected to the BeagleBoard over I2C interface to control motors. The software on Atmel micro-controller on the Trainer Board is responsible for receiving I2C commands and convert them to PWM signals which control the main car motor and steering servo.
- *StreamReceiver* is the callback interface which should be implemented by the cockpit application and used by vehicle to send compressed video frames as a chunks of binary data.

3.1.2 Video handling with Gstreamer

Vehicle application uses Gstreamer to perform video capturing, compression and handing the compressed stream over to Ice for transmission over the network. We are using Gstreamer's AppSink element to get access to the compressed and ready to transmit data. When the new chunk of date is available from the Gstreamer pipeline, the local callback function is invoked providing the pointer to the video data. In turn, vehicle application invokes remote callback interface to transmit the data over the network to the cockpit application.

Remote interface invocation is performed using Ice Asynchronous Method Invocation (AMI) which makes possible to collect more information about bufferization and transmission performance. This information is used to to trigger any QoS adaptation actions if necessary.

The following Gstreamer pipeline is used on the x86 Linux or Windows for development and testing purposes:

```
videotestsrc is-live=true ! video/x-raw-yuv,width=640,height=480,
framerate=30/1 ! videoscale name=qos-scaler !
capsfilter name=qos-caps
caps=video/x-raw-yuv,width=640,height=480 !
x264enc byte-stream=true bitrate=300 ! rtph264pay pt=96 !
appsink name=icesink
```

Videotestsrc element could be replaced with for example v4l2src to capture video from video camera. The following pipeline is used when we run vehicle application on the BeagleBoard and uses TI's h264 codec which runs on DSP and that is why can compress video in real-time on such low power device:

```
v4l2src always-copy=FALSE !
video/x-raw-yuv,width=320,height=240 ! ffmpegcolorspace !
video/x-raw-yuv,format=(fourcc)UYVY !
TIVidenc1 codecName=h264enc engineName=codecServer
bitRate=30000 genTimeStamps=TRUE byteStream=TRUE !
rtph264pay pt=96 ! appsink name=icesink
```

The used pipleline is not hard-coded but specified in the configuration file. It is very useful when experimenting on BeagleBoard because of long cross-compilation development cycle.

Since Gstreamer and Ice both have their own event loops we decide to run them in separate threads. Since both Gstreamer and Ice can also run multiple threads, it is very important to make proper synchronization between threads to prevent data corruptions and deadlocks. This is the most tricky part of the vehicle application.

3.2 Cockpit application

Cockpit application is intended to serve two main needs: a) receive and display sensor data such as for example video stream and b) capture and transmit control commands such as for example steering and acceleration from driver to the vehicle on-board software. Current version in particular supports video streaming and can transmit steering and acceleration commands. To issue control commands, keyboard and joystick could be used. From the software perspective, any joystick is visible as a set of axis and buttons. For example, steering wheel with pedals and a set of buttons is visible as a set of axis representing wheel and pedal angle. Buttons are sending notifications if pressed and released. This view makes it possible to support wide range of input devices such as for example conventional joysticks as well as game-oriented steering wheels with pedals. Based on our own experience, using steering wheel is the most convenient way to driver the car (which is somehow obvious :-)).

To implement requirements mentioned above, the following subsystems need to be implemented:

- *Communication* to receive sensor data and transmit control commands. Similar to the vehicle application, we are using Ice for communication purposes.
- *Visualization* to display received video stream, provide visual feedback when the driver issued control commands and display different kinds of textual messages for the driver. For this purposes we are using OpenGL together with certain windowing functionality available from SDL library.
- *Video decoding* to decode incoming video stream and prepare the raw frame data for displaying. We are using Gstreamer for video decoding purposes.
- *Input hardware support* to receive control commands from driver. Here we are relying on the corresponding functionality available in SDL library.

Integrating all the technologies mentioned above is not trivial. One of the big sources of complexity are so-called event loops. In particular, Ice required own event loop to to handle networking, Gstreamer requires own event loop to implement notification mechanism between decoding pipeline elements and SDL requires event loop to interface with windowing and operating subsystem. Each event loop is implemented as a blocking function which should be called after initialization and remains blocking until the application shut down. In our case, multiple event loops from different technologies should run simultaneously or be integrated with each other.

Each technology mentioned above provides certain mechanisms to integrate "foreign" event loops. However, such integration will require interventions at the very low level and either very complicated or impossible at all. That is why

we decide to run each event loop in own thread (or thread pool) and provide inter-thread communication infrastructure to exchange data between multiple subsystems. Figure 5 provides high-level overview of the cockpit subsystems and their interconnections.

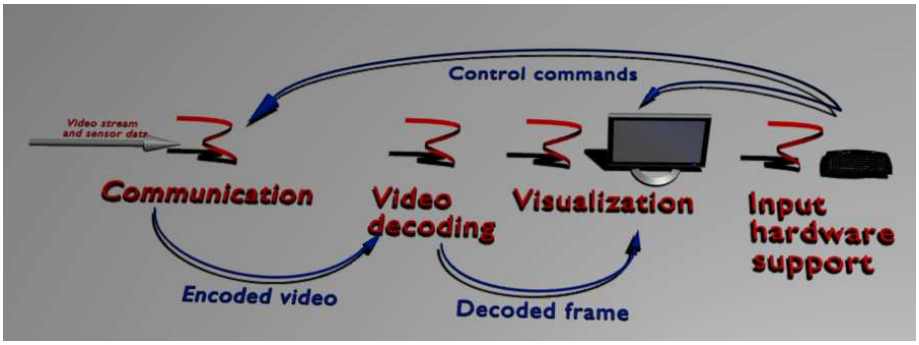

Figure 5: Subsystems in the cockpit application.

In addition to the SDL, Ice and GStreamer thread, we are using separate thread for connection establishment. This thread is periodically trying to obtain the reference to the remote vehicle object. If connection could not be established, then the thread sleeps for a couple of seconds and tries again.

When communication subsystem receives the chunk of data from the vehicle, it stores the data into thread safe queue which is also accessible by Gstreamer thread and video decoding subsystem. When decoding pipeline is running, we are receiving callbacks via user defined function when decoder needs more data. Our implementation of this function reads the next available data chunk from the queue populated by communication thread. We are using GStreamer's *appsrc* element to register required callbacks and supply video decoding pipeline with data.

As soon as the video decoding subsystem (Gstreamer pipeline running in the own thread) got enough data and the new frame is decoded, another custom callback function is invoked. We are using GStreamer's *fakesink* element to register our callback. The implementation of this function calls directly the visualization subsystem to notify about new frame availability.

The visualization subsystem (running in SDL thread) makes a copy of the new fame and sends the asynchronous event to trigger repainting process in the thread safe way. When event is dispatched, the video texture is updated with newly arrived data and drawn on the screen using corresponding OpenGL API. This drawing mechanism provides good performance and provides great flexibility for 3D artists to define the whole cockpit the way they like.

When the driver issues control command by means of keyboard or joystick, they are received by the SDL thread and made available for the application in form of events. These events are handled by corresponding callback functions. Received control data is then converted to the commands expected by vehicle. For example, key press events are counted and the desired acceleration and steering direction is calculated. This values are then transferred to the vehicle application using communication subsystem. In addition, the visualization sub-

system is notified to trigger repainting for on-screen objects (like steering wheel and tachometer) to provide the driver visual feedback for his actions.

The following subsections describe each particular subsystem with more details.

3.2.1 Communication subsystem

Cockpit application serves as a network client and server at the same time. When sending control commands, it acts as a client for the vehicle application. For this purposes, cockpit uses interfaces described in Section 3.1.1 and implemented by vehicle application. At the same time, to receive the video stream, cockpit application implements remote callback interface which is defined as following:

```
interface StreamReceiver {
  ["ami"] void nextChunk(ByteSeq chunk);
  idempotent QoSReport getQoSReport();
};
```

Vehicle application calls *nextChunk()* function every time the new data chunk becomes available from video encoder. When the data chunk is arrived over the network, communication subsystem makes this data available for the video decoding subsystem using special source element called *appsource*.

3.2.2 Visualization subsystem

There are several parts which need to be visualized in the cockpit application: static background, video frames, steering wheel, tachometer (indicated the acceleration requested by the driver) and notification area to display error messages and other types of information for the driver. For this purposes, we decided to create 3D model which consists of the objects which could be used to point where in 3D scene certain dynamic object should be positioned or could be manipulated to reflect current application state. In particular, the following objects are important for the cockpit application:

- *Video plane* is a square plane object and is used to place video frame on it as a texture.
- *Steering* is the 3D object which represents the steering wheel. This object is rotated around the central axis to provide the feedback when driver issued steering commands with keyboard or joystick.
- *Tachometer arrow* used to show how much acceleration is requested by the driver in the range of 0-100%. The arrow is rotated at run-time correspondingly.
- *Message area* is a square plane which is used to find the place in the 3D scene where to draw the message text. Text is drawn slightly raised (towards the viewer) and the plane plays the background role.

To define all the objects mentioned above, we are using Blender 3D modeler. Each object of interest has a name which is also known by the visualization

subsystem. At run-time, objects of interest are found in the whole model and then manipulated by the cockpit application.

We decide to use Wavefront OBJ file format which is widely accepted as the data interchange format between different 3D modeling applications. For this purposes we developed separate ObjReader library which is also freely available. Blender can export 3D geometry and materials in OBJ/MTL format.

3.2.3 Video decoding subsystem

Video decoding subsystem is responsible for building and running GStreamer video decoding pipeline and communicate with communication and visualization subsystem to receive compressed data and provide decoded video frames correspondingly. We are using *appsrc* element to feed the pipeline with data received by communication subsystem and *fakesink* to get access to decoded frames for visualization in 3D scene representing driver cockpit. Since GStreamer requires GLib's main loop running, we decide to dedicate the separate thread for it.

To makes it easier to experiment with different decoding pipelines, we read the actual pipeline from configuration file and using GStreamer's ability to build pipeline dynamically from textual definition. The following is one of pipelines we are currently using:

```
appsrc ! application/x-rtp, encoding-name=(string)H264,
payload=(int)96 ! rtph264depay !
video/x-h264 ! ffdec_h264 ! videoscale ! videorate !
video/x-raw-yuv, width=640, height=480, framerate=30/1 !
timeoverlay halign=right valign=top ! clockoverlay halign=left
valign=top time-format="%Y/%m/%d %H:%M:%S" ! ffmpegcolorspace !
video/x-raw-rgb, bpp=24, depth=24 ! fakesink sync=1
```

This pipeline receives h264 compressed video stream, makes appropriate color conversion, scaling and adds date and time as a video overlay. More pipelines and other cockpit configuration parameters could be found in the *misc/driverconsole.config* file. For example, it is possible to make the decoding pipeline more general with respect to used encoding algorithm by substituting *ffdec_h264* element with more general *decodebin2* element.

3.2.4 Input hardware support subsystem

To obtain input from the driver, keyboard and joystick is currently supported. Since most of the steering wheel and pedal devices are also viewed by the operating system as a joystick with corresponding set of axes and buttons, they are also supported. We are relying on the abstraction level and corresponding API provided by SDL to deal with hardware in the portable way.

In particular we are reacting on corresponding events sent by SDL to indicate that the user pushed the key, rotated the steering wheel or pushed the pedal. This events are processed to calculate the motor control commands which are represented as a desired duty of the corresponding actuator in percent. For example, to drive straight forward, i.e. position the forward wheels straight forward, the 50% of servo duty should be requested. Correspondingly 0% or 100% should be requested to make hard left or right turn.

The same scheme is used for acceleration. However, the percentage is interpreted slightly different by the vehicle on-board application. Duty from 0% to 50% is used to drive backwards where 0% is the fastest backwards speed and 51% to 100% is used to drive forward with 100% corresponding to maximum forward speed.