A Debugger Tool for Vision on Humanoid Framework

Aref Moqadam Mehr, Novin Shahroudi
Qazvin Islamic Azad University, Mechatronics Research Laboratory
Qazvin, Iran
{a.moqadammehr , n.shahroudi}@mrl-spl.ir

***Abstract*—In the RoboCup Standard Platform League (SPL), NAO biped robots are used for all teams in competitions. The robots have two on-board directional cameras and should perform fully autonomous, which requires precise data. The debugging tools always play critical role in developing reliable algorithms and calibrating sensors. In this paper we present a debugger and visualizer for vision of a standard platform league robot. This tool can be utilized for running off-line image processing algorithms aside calibrating the vision parameters like camera offsets and color lookup table. It also provides a very simple connection manager for transferring data with multiple robots and simulator.**

***Keywords*—vision; debugger; software; humanoid; robocup;**

I. INTRODUCTION

Obviously, a reliable debugging tool plays a significant role to accelerate the embedded software development process. Since resources are very limited on embedded platforms, it would be hard to track the algorithms process properly. Therefore visualizers, loggers, networking modules, etc. are mostly utilized to facilitate software developers. In RoboCup scenario, debugging tools and visualizers usually have additional features like having sensor calibrators. The debugging tools are provided as both integrated and separated applications. B-Human is one of the SPL competitors who introduce an integrated debugger in their code-release [3], [4]; it is a simulator for the game and robots, it contains tools for plotting or calibrating the sensor data and visualizing the data from each part too. B-Human debugger makes the possibility to create data log by using different macros in almost every part of the code. There are macros integrated for logging from numeric data, transmitting image, drawing shapes on image and plotting 2D and 3D ures. They achieved this aim by employing a modified version of Sim-Robot named B-Human Simulator that has capability of executing external widgets developed for providing extra features.

rUNSWift is another SPL team which has developed a debugger using C++-stream style [14] which is capable of dumping the robot memory in a file and/or connecting to a desktop client with a TCP/IP network connection using boost::asio C++ library. The debugger can be switched on or off during the running time. Further, to reduce the overhead of logging, all logs are written to volatile memory, to avoid large performance differences caused when data has to be synced to disk. Log files can be recovered from the memory over the network before the robot is switched off. A part of rUNSWift debugger is named off-Nao, which can run the vision module separately on the recorded frames for reproducing the problems and debug the vision module. Because separated debuggers are easier to develop, some teams prefer to use this approach. For instance, Berlin United Nao-TH and UPennelizers teams both use this method to debug their codes. This paper is organized to describe the implemented modules in MRL-SPL debugging tools. In Section 2, MRL-SPL framework is described. In section 3 networking approaches is demonstrated while section 4 is about sensor calibrations. Section 5 explains methods for constructing the color tables. In section 6, building log files and data serialization are discussed.

II. FRAMEWORK

NAO robot, manufactured by Aldebaran Robotics, is a bipedal robot with 25 degrees of freedom, two HD cameras with capability of capturing 72 FPS as main sensors, two infrared (IR) sensors, FSR, 3D gyro and accelerometer, and two ultra sonic (Sonar) sensors. It also possesses two speakers, and four microphones. The processor is an Intel ATOM Z530 1.6 GHz with 1 GB of RAM and a 2GB flash memory. This robot is also equipped with a wireless adapter with support of 802.11b/g/n [16]. NAO cameras can capture images every 33 milliseconds, which is about 30 images per second and in NAO-V4 it's possible to capture images simultaneously from both cameras. However for off-line image processing the image must be complete, for reviewing, there is no need to whole image. There are three methods employed to compress the image in order to send it on network or write it to a log file, during run time which described in section 7.

III. NETWORK

To get images from robot cameras and transfer them, there is a need to a network communication. There are various options. TCP and UDP and their derivatives are popular protocols used in almost every computer network program. Our requirements were to get images from robot's camera and upload or download files in some circumstances to/from robots easily and as fast as possible. Our team takes advantage of both TCP and UDP. Although we have implemented both, we use TCP in our Vision Debugger for its reliability and its rare packet loss. We employ Google Protobuf [5] in our debugging tools for serializing our data. Any changes in Protobuf packets means inability in retrieving on the other side. TCP guarantees both order and reception of all data. It is necessary to receive all pixels within an image for example. Instead UDP is best

fit to our team communication use to multi-cast the same amount of data to all teammates. Whether these packets may be received in-order or late is not very important and it is avoidable by filtering data and reducing rate of sending. Few teams have compared these two frequently used protocols in field of RoboCup [9] and Standard Platform League such as Austin Villa [10]. So we decided to compare them within our own implementation on two NAO robots. We use connection-oriented [15] design in TCP. An iterative server accepts requests in a new thread. It enables us to handle only one client at a time but with Non-blocking I/O and select [15] multiple connections are monitored and handled in the same working thread. In UDP an iterative server receives any data from specified address or port number in a new thread. The advantage of a separate thread is that all data are processed independently and the main process would not be affected for receiving delays. In our implementations server program runs on robot and debugger (the toolbox) is the client side. IP protocol and Maximum Transmission Unit (MTU) impose length of data in UDP and TCP packets. MTU is set by the hardware interface. For instance, MTU in Ethernet v2 is 1500 Bytes. As amount of data our Vision debugger and other toolboxes intend to transfer becomes larger, we need a proper approach to handle this issue. In TCP we choose the same approach as in [10]. In UDP, which is not a reliable protocol, we cannot follow the same approach for sending data as the initial packet containing size of the raw data can be lost. Our UDP module has a packeting solution to overcome this problem. Following structure is attached to each packet:

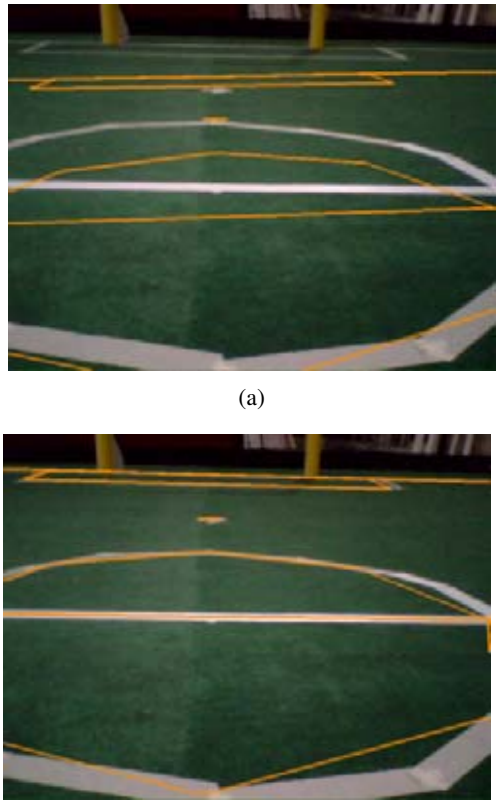
```
tag {
  number of all  packets
  number of this packet
  size   of all  packets
  size   of this packet
}
```

We split raw data into packets with reliable size before sending and attach this structure to each of them. Receiver side can extract every detail about the current reception such as number of all packets, remained packets to receive, total bytes to receive and remainder of that and consequently order of packets could be calculated easily. After all packets belonging to the same data received, the data is assembled.

As a conclusion we believe that TCP is the best choice for a debugging system such as our Vision Debugger. Packets reach to their destination as a whole in UDP [11]. So far it is a better choice But our data is large enough that transmitting them by UDP may cause packet loss. On the other hand TCP sends bytes as a stream which means any change or loss in bytes order is probable. This might seem as a negative point but TCP guarantees byte order by its built-in controls [11].

IV. CALIBRATION

The camera matrix is used for projecting objects onto the field. Robot dimensions together with the current position of the joints matrix relative to the torso are enough to determine the camera matrix. In addition to these parameters, the possibility of uncertain camera attachment together with



(a)

(b)

Fig. 1: (a) Non-Calibrated and (b) Calibrated images of the perceived lines

the probable lack of correspondence between torso vertical configuration and gyro zero position, enforce us to take also some robot-specific parameters into account. A small variation in the cameras orientation can lead to significant errors when farther objects are projected onto the field [3]. In order to calibrate the mentioned robot-specific parameters, there is a debug drawing that projects the field lines into the camera image as it can be observed in Fig. 1. This drawing is helpful for calibrating because the real lines and the projected lines only match if the camera matrix and hence the camera calibration are correct (assuming that the real position corresponds to the self-localization of the robot). The following paragraph will focus on camera calibration with correction of camera position matrix. Fig. 1 demonstrates manual camera calibration effects on the line perception performance. The observed misalignments of center circle and goal areas perceived lines in Fig. 1a are attributed to errors in the camera position and orientation matrices. The term ”Camera Parameters” is generally understood to mean two various assemblies: inherent and external. Inherent parameters are the essential part of camera characteristics such as lens distortion, focal length, etc., which can be determined easily by standard calibration routines presented in [6] or may be given by the manufacturer. Alternatively, external parameters are classified as extraneous and imposed constraints such as real time position estimation of head cameras relative to a specified reference point by transformation matrices [2].

Semi-Automatic Calibration

The fundamental idea of the semi-automatic calibration is to minimize error using Gauss-Newton algorithm [7]. The mean of error is calculated with the distance of the user selected points on the field lines in image with determined calibration lines. The user has to select some points in the image, on the field lines. The number of points user has to select is calculated experimentally during the several calibration tries. One of the benefits of this algorithm is wherever the points are, it can estimate the error and minimize it. It is useful in the competitions where the filed is usually crowded or some parts of the lines are covered by participants or robots.
The other feature of this method is that points can be chosen from all around the field by turning the robot's head. This is helpful for getting better result. We also provided an option for changing the robot position so that robot calibration does not depends on a fixed placement.

V. Color Lookup Table

A fast identification of color classes is achieved by pre-computing the class of each color and storing it in a color lookup table [4] (or simply color table). Calibrating color table is another item for robot calibration and preparation before a game. There are several patterns to fetch data from image; these patterns are called Color Spaces. YUV, HSL and RGB are all color spaces used in computer vision, from which YUV performs better in color-coded environments like RoboCup fields, also in encoding the camera outputs. However reducing the certainty parameters from 256 colors to 126 or 64 colors might make an uncertain color table, the less memory usage of it worth the uncertainty. This gets important when the resources are limited like in NAO robots. It also could result in a better color table, which is easier to calibrate as well as having almost the same performance in classifying images.

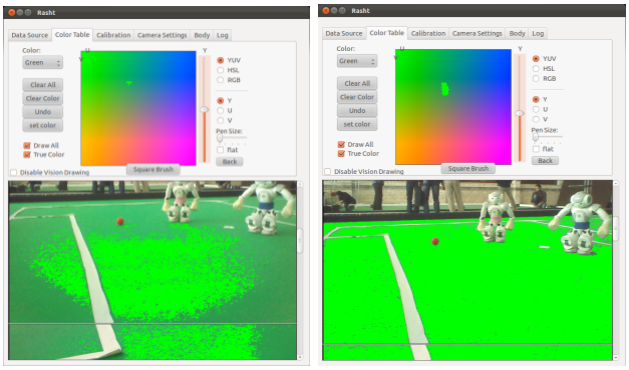


Fig. 2: (left one) poor color table, (right one) rich color table.

Defining a range for each color classes could be easier than inserting all the colors directly by selecting them. The color ranges could be defined in polar based color spaces, which have a value to specify the colors (theta) like Hue in HSL color space. Selecting colors by ranges is an approach to a dense color table resulting more reliable object detection. The purpose of HSL color table is to select colors which have small amount of pixels in image, like white colors which are mainly used to detect the field lines but hard to set via brushes. This could be observed in Fig. 3.

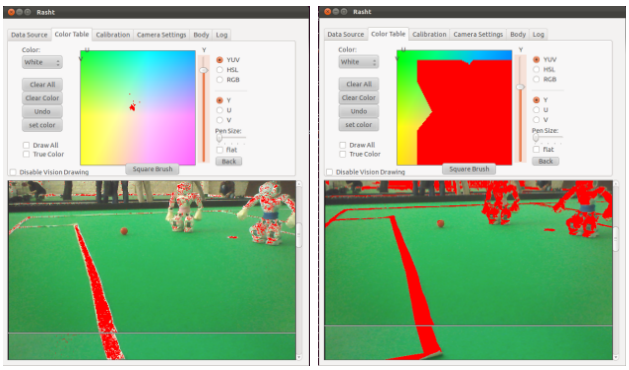


Fig. 3: differences between YUV (left one) and HSL (right one) color tables.

In order to use the color table in robot the selected HSL color ranges must be converted to YUV space. Each HSL color is converted to YUV using RGB as a medial color space. The conversion formulas are shown in equation 2 and 3 respectively.

$$C = (1 - |2L - 1|) \times S_{HSL}$$

$$H' = \frac{H}{60^\circ}$$

$$X = C(1 - |H' mod 2 - 1|)$$

$$(R_1, R_2, R_3) = \begin{cases} (C, X, 0) & if 0 \leq H' < 1 \\ (X, C, 0) & if 1 \leq H' < 2 \\ (0, C, X) & if 2 \leq H' < 3 \\ (0, X, C) & if 3 \leq H' < 4 \\ (X, 0, C) & if 4 \leq H' < 5 \\ (C, 0, X) & if 5 \leq H' < 6 \end{cases} \quad (1)$$

$$m = L - \frac{C}{2}$$

$$(R, G, B) = (R_1 + m, G_1 + m, B_1 + m)$$

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.436 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (2)$$

VI. Controller

To control robot locomotion and head motion a controller has been implemented in this debugger, which could be helpful during calibration or running a fake game. This controller sends walk command or the head motion pattern to robot by TCP/IP, which is previously discussed. These commands include the walking velocity in three dimensions (X, Y, theta) and head motion command that can be tracking the goal or the ball. This is achieved by developing scripts for head motion and walk in robot and switching them by user.

VII. LOGGING DATA

In order to reproduce the problems that rarely occur, for debugging, the robot sensor data must be written to a log file. Google Protobuf [5] is being employed to use as a protocol buffer with the same structure used to transmit data on network. The Serialization and De-Serialization functions are implemented in each representation class to be used as an interface for Protobuf. These functions convert objects to Protobuf classes and conversely.
Google Protobuf has methods for converting the objects to an array of data, which can be sent on network or stored in a file. We use a pattern to save data in file, just like sending it via network. A number indicating the packet size is added to the beginning of data in order to surf the log file. For compressing image, only Y channel of YUV color space is picked, which represents the lightness of that pixel so that it could provide a gray-scale image. The image captured by cameras is in YUV422 color space, which means there is only one Y value for each pixel. It also decreases the size of each channel from 1 byte to 4bit (half a byte) that consequently reduces the image size to half. Another method for compressing the image is to reduce its resolution, so in each set of *n* pixels only one of them is picked and the others are skipped. The Equation 3 represents how to compress the image in which Img_{out} will be transmitted on network and Img_{raw} is the image captured by camera. *W* is a constant for width of the image and i_y and i_x are respectively position of the $i-th$ pixel in height and width (X and Y vector).

$$Img_{out}[\frac{i_y * W}{8} + \frac{i_x}{2}] = Img_{raw}[\frac{i_y * W}{2} + i_x] \qquad (3)$$

VIII. OFF-LINE IMAGE PROCESSING

Running Image Processing code on the off-line (recorded) log file enables user to debug faster by making debugging instructions easy and also make the possibility to run Image Processing for one single frame over and over, in order to solve problems for special cases. An object from Image Processing module has been created which provides the image as the camera-captured image by reading from the log file. It creates Image Processing module as an external widget, which can be run by simulating the environment and sensors. The connection between the debugger and the image processing module is same as the connection used to transmitting data in network. The widgets and the main form are connected using Qt signal/slot approach.

EXPERIMENTAL RESULTS

In our first demonstration (Fig. 5a) the fact that UDP is faster than TCP is proved but it does not mean UDP packets will get to their destination earlier than TCP (Fig. 5b). It is only fast in making packets in the origin and destination. This test is taken place in a local machine. As the time takes a packet gets to its destination on a local machine becomes very short the calculated time is the amount spent for creation and reading it.

As it is mentioned in [11] TCP for its reliability and UDP for its fast and easy transmission are both good choices. Our demonstrations can help to see where and when each of them can be used, especially in SPL. Round Trip Time (RTT) is the main criterion in these tests, which is computed for both protocols. Ping command is also used as a valid criterion for RTT calculation. We did two different tests in a single hop wireless network on two NAO robots. First was testing the consistency of UDP and TCP by measuring the RTT of a 500 bytes packet in 30 different tests named test steps in Fig. 5a. By consistency we mean how many times the result is similar with other tests. Table-I shows the average RTT calculated for each method we used in low and high traffic networks.

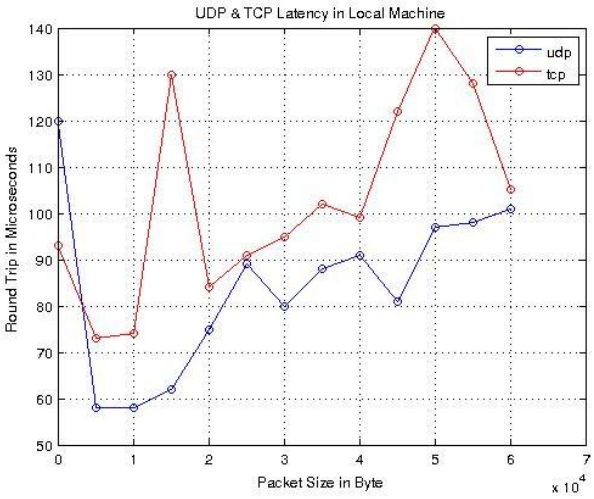


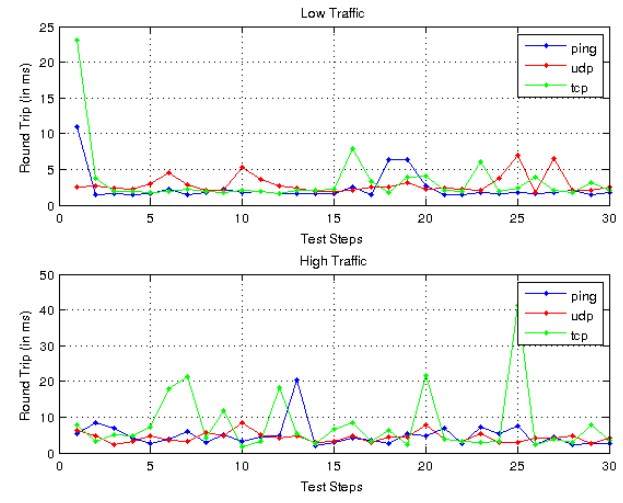
Fig. 4: Comparing UDP and TCP speed in local machine

Low Traffic		High Traffic	
Method	RTT	Method	RTT
Ping	2.3620 ms	Ping	4.9173 ms
UDP	2.8914 ms	UDP	4.3063 ms
TCP	3.3539 ms	TCP	7.9125 ms

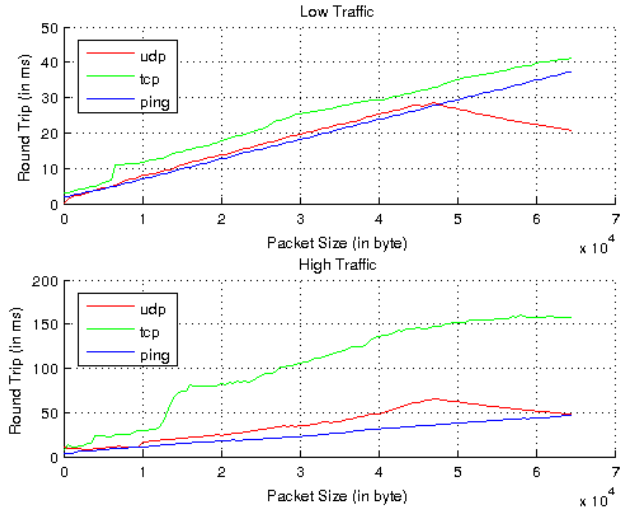
TABLE I: Average Round Trip Time in low and high traffic conditions for fixed packet size

In our second demonstration we increased size of the packets incrementally to see how each of UDP and TCP changes its behavior by increasing the size of packet. Again this test took place in both a low traffic and a congested network. We added this traffic by copying a large file from one computer to another in the network and a GameController [13] running which was broadcasting all the time. Fig. 5b shows the moving average plot of this demonstration. Zero values indicate the packet loss.

From the above figures it could be understood that TCP has a larger Round Trip Time in packets larger than 10Kbytes but has almost the same time with UDP in lower sizes. In Fig. 5 UDP packets are lost after 46500 Bytes. This happened because the maximum size of buffer set as an option for the socket could not go further on robots.



(a)



(b)

Fig. 5: Comparing RTTs calculated from Ping, UDP and TCP with (a) fixed and (b) incremental packet size in low and high traffic network.

ACKNOWLEDGMENT

Authors gratefully acknowledge the technical support of Mechatronics Research Laboratory and MRL-SPL team members, for their hard work during the development of our team for this years RoboCup competitions and also a special thanks to Mr. Mohammad Shafiei, head of vision group in MRL-SPL and Mr. Mohammad Ali Zakeri for their valuable suggestions and reviews about this paper.

REFERENCES

[1] Nokia Development Team, Digia, Qt-SDK-4.7, doc.qt.nokia.com

[2] E. Hashemi, M. Ghaffari Jadid, M. Lashgarian, M. Yaghobi, M. Shafiei R. N. "Particle Filter Based Localization of the Nao Biped Robots" Fauclty of Industrial and Mechanical Eng., Faculty of Electrical, IT, and Computer Eng., Qazvin Branch, Islamic Azad University, Qazvin, Iran.

[3] T. Rfer, T. Laue, "B-Human Team Report and Code Release 2011", Deutsches Forschungszentrum fr Knstliche Intelligenz, Enrique-Schmidt-Str. 5, 28359 Bremen, Germany Universitt Bremen, Fachbereich 3, Postfach 330440, 28334 Bremen, Germany , November 3, 2011

[4] T. Rfer, T. Laue, "B-Human Team Report and Code Release 2010", Deutsches Forschungszentrum fr Knstliche Intelligenz, Enrique-Schmidt-Str. 5, 28359 Bremen, Germany Universitt Bremen, Fachbereich 3, Postfach 330440, 28334 Bremen, Germany , October 1, 2010

[5] Google: Google protocol buffers Protobuf (July 2008) Open source project. https://code.google.com/apis/protocolbuffers/docs/overview.html

[6] Z. Zhang, "A flexible new technique for camera calibration," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 11, pp. 13301334, 2000.

[7] F. Dan Foresee and Martin T. Hagan, "GAUSS-NEWTON APPROXIMATION TO BAYESIAN LEARNING", Lucent Technologies , Oklahoma City, OK

[8] J. Postel, "Internet Protocol: DARPA Internet Program Protocol Specification," RFC 791, Sept. 1981.

[9] P. Stone, K. Dresner. "The UT Austin Villa 2005 RoboCup Four-Legged Team. Technical Report" UT-AI-TR-05-325, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, 2005.

[10] P. Stone, K. Dresner. "The UT Austin Villa 2003 RoboCup Four-Legged Team. Technical Report" UT-AI-TR-03-304, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, 2003.

[11] W. R. Stevens, UNIX Network Programming Vol. 1, Third Edition: The Sockets Networking API, Addison Wesley, 2003.

[12] "RFC 1122 - Requirements for Internet Hosts – Communication Layers". p.42. Retrieved 2012-03-19.

[13] Robocup Committee "Robocup Game Controller" http://www.tzi.de/spl/pub/Website/Downloads/GameController2012.zip

[14] A. Ratter , B. Hengst, "rUNSWift Team Report 2010", School of Computer Science & Engineering, University of New South Wales, Sydney 2052, Australia, October 30, 2010

[15] IBM, iSeries Socket programming version 5 release 3. RFC793 , p. 13

[16] Aldebaran Robotics, "Data sheet NAO Humanoid Next Gen-H21/H25 Model", November 2012