NUbugger: A Visual Real-Time Robot Debugging System

Brendan Annable, David Budden, and Alexandre Mendes

School of Electrical Engineering and Computer Science,
Faculty of Engineering and Built Environment,
The University of Newcastle, Callaghan, NSW, 2308, Australia
{brendan.annable,david.budden}@uon.edu.au,
alexandre.mendes@newcastle.edu.au

Abstract. As modern autonomous robots have improved in their ability to demonstrate human-like motor skills and reasoning, the size and complexity of software systems have increased proportionally, with developers actively working to leverage the full processing performance of next-generation computational hardware. This software complexity corresponds with increased difficulty in debugging low-level coding issues, with the traditional methodology of inferring such issues from emergent high-level behaviour rapidly approaching intractability. This paper details the development and functionality of NUbugger: a visual, real-time and open source robot debugging utility that provides the user with comprehensive information regarding low-level functionality. This represents a paradigm shift from corrective to preventative debugging, and concrete examples of the application of NUbugger to the identification of fundamental implementation errors are described. The system implementation facilitates simple and rapid extension or modification, making it a useful utility for debugging any similar complex robotic framework.

Keywords: debugging, robotics, open source, visualisation.

1 Introduction

The problem of developing a team of humanoid robots capable of defeating the FIFA World Cup champion team, coined "The Millennium Challenge" [7], has been a milestone that has driven research in the fields of artificial intelligence, robotics and computer vision for over a decade. Corresponding with the continual improvement in a robot's ability to demonstrate human-like motor skills and reasoning is an exponential blowout in software size and complexity, facilitated by the evolution of robot platforms and subsequent advances in processor performance (from the 384 MHz RISC-based processors of the Sony AIBO ERS-210 (2002) to the 1.6 GHz Intel Atom processors of the Robotis DARwIn-OP [5] platform (2012)); a trend often inferred from Moore's Law [11].

As with any system of software or hardware, exponential increases in system size and complexity necessitate the introduction of hierarchial layers of abstraction, allowing low-level functionality to be handled transparently by higher-level

S. Behnke et al. (Eds.): RoboCup 2013, LNAI 8371, pp. 544–551, 2014.

functions, classes and packages. As the majority of open source libraries are supported by large developer communities, one critical factor is often ignored: the cascading effect of low-level errors, and the difficulty of locating the source of such errors by qualitative observations of emergent system misbehaviour (such as a robot refusing to kick a ball [2]). The probability of such low-level issues is increased by a number of factors encountered in typical RoboCup research environments: few team members handling a large number of issues, strict deadlines and the disjoint nature of development (both in terms of frequent developer turnover and one-developer-per-subsystem strategies [9]).

An analogy may be drawn between complex robotic systems and modern genomics, where abnormalities in an organism's genotype are almost exclusively identified by complex emergent behaviour (e.g. a disease or hereditary trait) at the phenotypic level. Although determining the root cause of some observed abnormalities is a straightforward process, more complex system misbehaviour may be the result of a significantly larger number of low-level contributing factors. Due to the massive complexity of biological systems and the inability to make perfect observations at the genome-level, the exact systematic causes of many common diseases remain largely unknown, despite years of research by very large teams.

Although robotic and biological systems are (to date) fundamentally different in implementation, they exhibit two primary common traits: massive complexity and high-level issues caused by nontrivial combinations of low-level implementation errors. In this sense, the above analogy provides a useful insight to common debugging methodologies; locating low-level issues by observing high-level behaviour is an increasingly intractable problem. However, unlike in biological systems, where network models are constructed and analysed to infer genotype from phenotype, the "genome" of a robotic system (i.e. the low-level software behaviour and environmental response) is directly observable in real-time. Specifically, the following debugging methodologies are possible:

- **Data Logs:** Most robotic systems provide functionality for generating debug logs, commonly enabled by setting the value of some *debug verbosity* parameter at compile-time (such as per the NUbots system [9]). Once an error has been appropriately reproduced, data logs are generated capturing some of (and not limited to) the following information as a function of time:
  - Sensor values (accelerometer, gyroscopes, pressure sensors, etc.)
  - Servo positional values (from which kinematics and pose may be inferred)
  - Current captured image, colour-classified image [1] and recognised features
  - Self-localisation belief and perceived location of dynamic features (such as the ball in robot soccer [2])

  This information may be later analysed (either manually or by applying classification techniques for well-known errors) to discover the sources of error.
- **Visual Monitoring:** Although a significant improvement over high-level inferential debugging, data logs have three major shortcomings: limited

memory resources restricting the amount or resolution of data that can be collected; the requirement for an error to be reproducible, allowing it to be captured in a log once debugging has been enabled; and the adoption of a purely *corrective* (rather than *preventative*) debugging methodology. All of these issues are addressed by a *visual monitoring* methodology, where all critical system information is streamed in real-time to a graphical web client, to be monitored by the user. Concretely, this addresses the aforementioned issues in the following ways:

- By comparison to memory, network communication is relatively cheap. Large amounts of information (including real-time video) may be streamed from a robot to a corresponding web client for an effectively unlimited period of time.
- Uncommon issues may be immediately identified by the user, removing the presumption of error reproducibility.
- Abnormalities in low level functionality (such as an incorrectly classified image or reduced video frame-rate) may be identified and corrected before they are able to visually affect high-level behavioural performance.

This paper describes the implementation of *NUbugger* (Newcastle University's Debugger); a visual, real-time and open source robot debugging utility that addresses the aforementioned issues. Firstly, an overview of the system implementation is provided, both in terms of high (system inputs, outputs and architecture) and low-level (languages and library dependencies) application structure. The main functionality of NUbugger is explained, with concrete visual examples of the real-time information provided to the user. Finally, the significance and outcomes of the system are justified by providing examples of actual low-level errors in the NUbots RoboCup source code, that have been identified and corrected as a direct result of NUbugger's implementation.

2 Implementation

NUbugger implements a many-to-many service between robots and web clients, via a single web server. Concretely, an arbitrary number of robots (3-4 in the case of RoboCup humanoid league soccer) are able to stream real-time, low-level system information wirelessly to a single web server (which may be a robot in itself). This information is then distributed to an arbitrary number of web clients, allowing users to monitor performance-critical visualisations of sensory data and emergent high-level behaviour. This process is illustrated in Fig. 1.

Due to the complexity of streaming large quantities of data from a robot (implemented in C++) to web server (implemented in JavaScript), and finally to a lightweight web client for real time display, a number of libraries were utilised to provide abstraction over low-level networking and visualisation mechanisms. The following libraries facilitated rapid application development, in addition to minimising the effort required for future extension or modification:

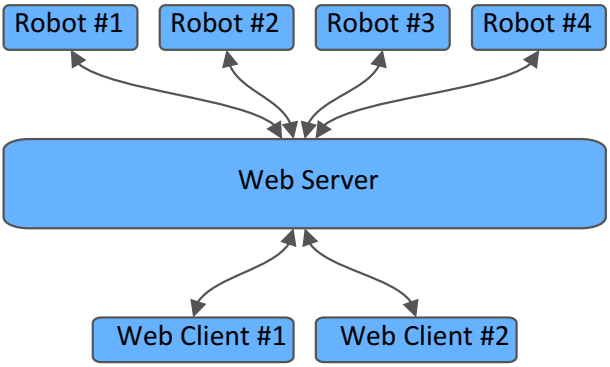

Fig. 1. Visual representation of the many-to-many NUbugger implementation, allowing multiple users to visualise the real-time performance of an arbitrary number of robots. The selected libraries provide sufficient levels of abstraction over underlying network communication that the robots and web clients need not be coded in the same language.

- **ØMQ:** Referred to as "the intelligent transport layer" by developers iMatrix, ØMQ is a high-performance, asynchronous message library facilitating speed and control over low-level message passing [6]. It was chosen for implementation of the raw transport layer from robot to web server due to the ease of embedding into a pre-existing application; concretely, it provides direct support for C++, in addition to providing simple abstractions over multithreading and automatic reconnection.
- **Protocol Buffers:** Developed by Google, Protocol Buffers are a language-independent, cross-platform and extensible mechanism for serialising structured data [4]. As ØMQ only provides a transportation mechanism for raw binary data, such structures are critical for the addition of application level information. Protocol Buffers allows for objects to be created and transported across the network in a packed binary form, rather than an inefficient ASCII representation (utilised by JSON or similar transport protocol alternatives). Language independence is realised in a manner that allows C++ structures to be transparently interpreted by the JavaScript browser client.
- **Socket.IO:** Developed by Guillermo Rauch, Socket.IO is a JavaScript library for real-time web applications [10], and was implemented as the raw transport layer from web server to web client. Socket.IO was chosen due to its native support for WebSockets (the only streaming technology natively supported by most modern web browsers), in addition to its transparent support of automatic reconnection and fall-back transport methods. As not all transports supported by Socket.IO support binary, Base64 encoding was applied.
- **Express:** Developed by VisionMedia, Express is a minimal and flexible node.js web application framework, providing a robust set of features for building single, multi-page and hybrid web applications [13]. Express was chosen for web server implementation due to easy integration with the

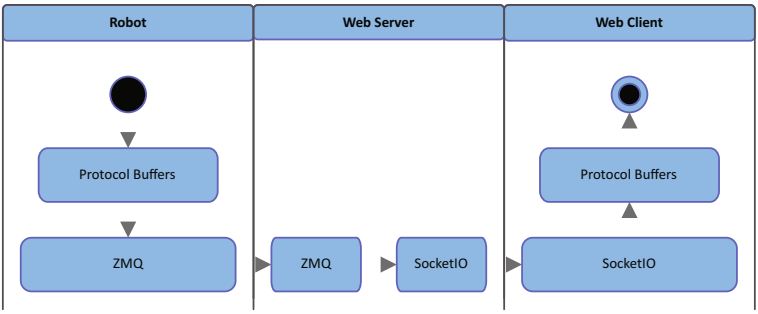

Fig. 2. Visual representation of the flow of information between the various networking packages implemented during the development of NUbugger. The selected libraries provide sufficient levels of abstraction over underlying network communication that the robots and web clients need not be coded in the same language.

JavaScript web client, in addition to its low computational expense, which allows for the option of running the web server directly on the robot (if other options are not available).

- **Three.js:** Developed by Ricardo Cabello, Three.js is a lightweight, cross-browser JavaScript library that allows animated 3-dimensional computer graphics to be displayed directly within a web browser, by providing an abstracted interface over WebGL [3]. It was implemented for the 3D rendering of the main application display.
- **Ext JS:** Developed by Sencha, Ext JS is a pure JavaScript application framework for building interactive web applications, using techniques such as Ajax, DHTML and DOM scripting [12]. It provides several widget and component templates that are commonly used within a web development context, and was used for the implementation of movable, resizable and configuration windows to contain each developed application user interface. Ext JS provides flexibility for future extensions to the user interface by allowing for the simple addition of further displays.

The flow of information between the various networking packages is illustrated in Fig. 2.

3 Functionality

As demonstrated in Sec. 2, the NUbugger implementation allows for critical system information to be streamed in real time from robot (in this case the Robotis DARwIn-OP [5]) to web client (via web server). Although readily modifiable and extensible, the NUbots debugging environment currently consists of the following elements:

- The main display, as demonstrated on the right half of Fig. 3. This interface provides a visualisation the robots self-localisation belief (i.e. the position

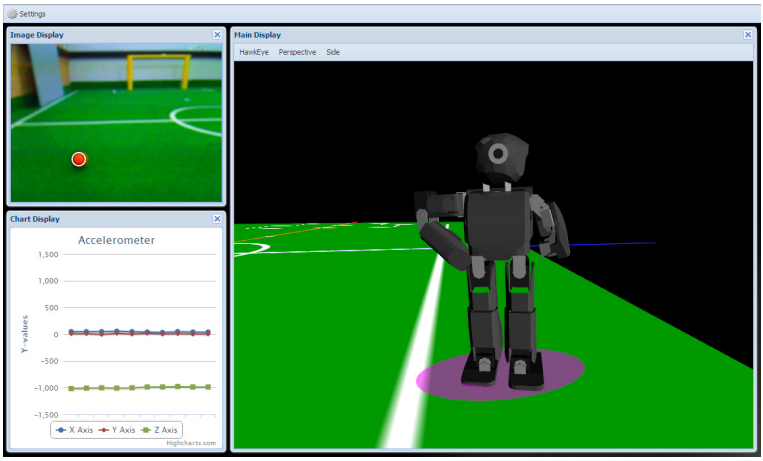

Fig. 3. The NUbots configuration of the NUbugger utility, demonstrating: the main display (right), image display (top-right) and scrolling chart display (bottom-left). These displays are able to be customised by the user to provide any manner of real-time information from the robot.

and orientation of the rendered 3D model [8]) and error (indicated by a transparent purple ellipse), in addition to the current pose of the robot, which utilises real-time accelerometer and servo positional data.

- An image display, as demonstrated on the top-left of Fig. 3 and Fig. 4b. This interface provides an indication of what the robot sees in real time, and includes a number of overlays to allow the user to visualise image classification [1] or detected salient features [2].
- A scrolling chart display, as demonstrated on the bottom-left of Fig. 3 and Fig. 4a. This interface is able to provide real-time data directly from any number of the robot's sensors, including: accelerometers (as demonstrated), gyroscopes, pressure sensors and temperature monitors.

4 Conclusion

In the short time since its development, NUbugger has already been applied with great success to the University of Newcastle's NUbots RoboCup team [9]. Concretely, it has assisted with the following low-level issues:

- **Video latency:** A previous version of the NUbots vision system maintained a buffer of 20 image frames [9], facilitating random access and explicitly enforcing thread-safe execution. Recent redevelopment of the vision system removed the need for this functionality [2], and introduced conflicts preventing buffered frames from being accessed synchronously with sensor data. This resulted in up to a 0.6 second latency in image processing; a significant

(a) (b)

Fig. 4. Two examples of NUbugger debugging interfaces: a) the scrolling chart display, demonstrating real-time information from the robot's 3-axis accelerometer; and b) the image display, with an overlay demonstrating the correct identification of the ball [2].

issue which went undiscovered until streamed in real-time to the NUbugger utility, despite causing an observable reduction in self-localisation accuracy and overall system performance.

- **Ball detection accuracy:** An implementation error in the vision system prevented the robot's head pitch and yaw from being considered when projecting ball-localisation coordinates from image to field-plane, causing significant inaccuracy whenever the robot was not facing directly ahead. This was identified in the main NUbugger display, which provides an overlay of the robot's ball positional belief; rotating the robot's head from side-to-side caused the overlay to transcribe an arc about the robot, despite the fact that the ball remained physically stationary.
- **Camera firmware:** The Robotis DARwIn-OP robot platform is equipped with a Logitech C905 camera, which utilises the Linux UVC driver [5]. Although this driver provides control over a large subset of fundamental camera parameters (such as brightness and contrast), a small number of parameters (including auto white balance and a number of proprietary Logitech colour correction values) remain inaccessible. Although this is a known and unresolved issue, NUbugger allows for instant detection of one of the resultant errors: automatic white balance adjustment caused by lighting variations across the soccer field, resulting in a dramatic reduction in classification accuracy and therefore object recognition performance. Once the white balance of the streaming images is observed to have changed, the robot may be instantly substituted and restarted.

Although still in the process of active development, it is hoped that NUbugger will be successfully adopted by other RoboCup teams. This may be possible at a number of levels:

- The adoption of the complete NUbugger system (both server and client applications), with modifications made to the server-robot interface where necessitated by different hardware platforms.

– The adoption of the NUbugger server application, with the functionality and layout of the client modified to suit an individual team's specific needs.

In recent months, it has proven critical to the identification of low-level issues that were plaguing system performance, but near-impossible to identify using traditional debugging methodologies. The latest NUbugger source code is available for download at `https://github.com/nubots/NUbugger`, and the authors are happy to collaborate with other RoboCup teams in the extension of the framework to better incorporate their needs.

References

1. Budden, D., Fenn, S., Mendes, A., Chalup, S.: Evaluation of colour models for computer vision using cluster validation techniques. In: Chen, X., Stone, P., Sucar, L.E., van der Zant, T. (eds.) RoboCup 2012. LNCS (LNAI), vol. 7500, pp. 261–272. Springer, Heidelberg (2013)
2. Budden, D., Fenn, S., Walker, J., Mendes, A.: A novel approach to ball detection for humanoid robot soccer. In: Thielscher, M., Zhang, D. (eds.) AI 2012. LNCS, vol. 7691, pp. 827–838. Springer, Heidelberg (2012)
3. Cabello, R.: Three.js (2013), `https://github.com/mrdoob/three.js/`
4. Google: Protocol Buffers (2012), `https://developers.google.com/protocol-buffers/`
5. Ha, I., Tamura, Y., Asama, H., Han, J., Hong, D.: Development of open humanoid platform DARwIn-OP. In: Proceedings of SICE Annual Conference, SICE 2011, pp. 2178–2181. IEEE (2011)
6. iMatrix: ØMQ: The Intelligent Transport Layer (2013), `http://www.zeromq.org/`
7. Kitano, H., Asada, M.: The robocup humanoid challenge as the millennium challenge for advanced robotics. Advanced Robotics 13(8), 723–736 (1998)
8. Michel, O.: Webots: Symbiosis between virtual and real mobile robots. In: Heudin, J.-C. (ed.) Virtual Worlds 1998. LNCS (LNAI), vol. 1434, pp. 254–263. Springer, Heidelberg (1998)
9. Nicklin, S.P., Bhatia, S., Budden, D., King, R.A., Kulk, J., Walker, J., Wong, A.S., Chalup, S.K.: The nubots team description for (2011)
10. Rauch, G.: Socket.IO, `http://socket.io/` (2012)
11. Schaller, R.R.: Moore's law: past, present and future. IEEE Spectrum 34(6), 52–59 (1997)
12. Sencha: Sencha Ext JS: JavaScript Framework for Rich Desktop Apps. (2013), `http://www.sencha.com/products/extjs`
13. VisionMedia: Express: Web application framework for node (2013), `http://expressjs.com/`