Aquila 2.0
Software Architecture for Cognitive Robotics

Martin Peniak, Anthony Morse and Angelo Cangelosi
Centre for Robotics and Neural Systems
Plymouth University, Plymouth, PL4 8AA, United Kingdom
Tel: +44-1752-586217, Fax: +44-1752-586300

Abstract—The modelling of the integration of various cognitive skills and modalities requires complex and computationally intensive algorithms running in parallel while controlling high-performance systems. The distribution of processing across many computers has certainly advanced our software ecosystem and opened up research to new possibilities. While this was an essential move, we are aspiring to augment the field of cognitive robotics by providing Aquila 2.0, a novel hi-performance software architecture utilising cross-platform, heterogeneous CPU-GPU modules loosely coupled with GUIs used for module management and data visualisation.

I. INTRODUCTION

Cognitive robotics is concerned with endowing robots with high-level cognitive capabilities to enable the achievement of complex goals in complex environments. These capabilities include perception processing, attention allocation, planning, anticipation, and reasoning about their own and other agents mental states.

One of the most popular cognitive robotic platforms is the iCub (www.icub.org) humanoid robot, which is a true achievement when it comes to the number of joints that can be controlled (53 deg. of freedom) and the variety of different sensors that can be utilised (vision, sound, skin, touch sensors, force-torque sensors, position sensors, gyroscopes). Dealing with so much data while running software designed to make sense of it has been a challenge for years. Fitzpatrick and colleagues have developed Y ARP (Yet Another Robotic Platform), which addressed this issue, standardised the software development and encouraged platform-independence, modularity, scalability and reuse via location-independent processes that can independently communicate via hardware-nonspecific protocols.

Cognitive robotics community is highly multi-disciplinary field of research involving computer scientists, psychologists, neuroscientists, linguists etc. We have been working with a variety of students and researchers from all of these fields and we feel that one of that major obstacles hindering progress in cognitive robotics community is the lack of tools facilitating rapid development, integration and visualisation of modules. Some are simply looking to use certain features of the iCub (e.g. record vision, sensorimotor movements, control objects in the simulator) while others would like to develop their own modules with graphical user interfaces (GUIs) and run them in any number of instances anywhere across the network while centralising their management and data visualisation on a host machine. The technical challenges involved in this, however, discourage many from embarking on such development.

This paper presents Aquila 2.0, an open-source, cross-platform software architecture that addresses these issues and was developed by researchers for researchers, students and enthusiasts who do not necessarily want to spend days writing new modules or integrating them together to create higher-level modules or novel cognitive models. Aquila software architecture makes use of independent, hi-performance modules that can run anywhere across the network in any number of instances and using any number of available GPU (Graphics Processing Unit) devices. These modules are loosely coupled with their GUIs centralised and dynamically generated by Aquila. Modules can spawn other modules on any number of machines and GPU cards available on the network, which allows running complex, computationally demanding tasks that would not have been previously possible or feasible.

II. MOTIVATION

The modelling of the integration of various cognitive skills and modalities requires complex and computationally intensive algorithms running in parallel while controlling high-performance systems. The processing requirements are increasing with every added feature and it is not uncommon that at the end of the software development stage a particular system is unable to cope with fast-response robot-control tasks. Y ARP provides means for distributing any number of processes across any number of machines using any of the available underlying communication protocols. Y ARP has become a standard for developing and interconnecting modules.

Distributing processing across many computers has certainly advanced our software ecosystem and opened up research to new possibilities. While this was an essential move, we are aspiring to augment the field of cognitive robotics by providing Aquila 2.0, a novel hi-performance software architecture utilising cross-platform, heterogeneous CPU-GPU modules loosely coupled with their GUIs used for module management and data visualisation.

Around the year 2003, to overcome the energy consumption and heat-dissipation problems of standard PC processors, manufacturers started to produce computers with multiple cores. In the meanwhile, manufacturers have been looking into new technologies that would increase the number of transistors per wafer. However, reducing these dimensions comes at a price since the current leakage becomes a problem.

Since 2003, the production of semiconductors has been divided into multicore and manycore design trajectories. Manycore design aims to increase the processing power by increasing the number of cores in a processor. This number was...
doubling with each semiconductor process generation starting with dual-core chips and reaching hyper-threaded hexa-core systems. A manycore system is fundamentally different with regards to its design philosophy. While CPUs are optimised for the processing of sequential code and feature sophisticated control logic and large cache memories, the GPU design philosophy emerged from the fast growing video industry where massive numbers of floating point operations are required to render every single frame. As a result, a GPU chip has most of its area dedicated to processing of the floating point operations and features only tiny cache memories.

In 2006, NVidia released GeForce 8800 GPU, which was capable of mapping separate programmable graphics processes to an array of GPUs, which paved the way to first general purpose computing using parallel GPU processors. GPGPU was an intermediate step where graphics card programmers had to use the OpenGL or DirectX API to implement their programs. Using the GPGPU technique many different applications have achieved dramatic speed improvements.

Parallel computing using GPU devices is being increasingly taken up by industry and academies. Many commercial and research applications have migrated from using solely standard CPU processors to a heterogeneous CPU-GPU environments where each architecture does what is best at. Most of these applications achieve tremendous speed-ups in performance. Since quantum computing is still in its infancy and CPUs are approaching the processing limits constrained by the physical laws, it seems that heterogeneous CPU-GPU parallel computing is the next paradigm, which we would like to support and fully utilise in Aquila 2.0.

III. Architecture

Aquila was written entirely in C++ object-oriented language, which was motivated by the fact that C++ is well supported and widely used by many developers worldwide. NVIDIA CUDA (Compute Unified Device Architecture) was chosen for the development of GPU-accelerated functions. CUDA is well-written and documented, widely adopted and supported by an install base of over 300 million CUDA-enabled GPUs in notebooks, workstations, compute clusters and supercomputers. Aquila is based on Qt, a leading development framework designed for creating applications and user interfaces for desktop, embedded and mobile platforms. YARP enables Aquila modules to communicate with their GUIs as well as other modules and devices. CMake is used to control software compilation process using simple platform and compiler independent configuration files. All of the above frameworks, libraries and tools are cross-platform and free to use.

Once of the most important aspects and contributions of Aquila lies in its fundamental architectural design, which makes a clear division between modules that do the actual work and their GUIs that provide easy-to-use controls over modules and the possibility to visualise their behaviour. The module GUIs are compiled as part of Aquila GUI, which is able to dynamically add module GUIs under new tabs. A user can then use and manage these modules, see their visualisations, duplicate or close them just as one would in a typical web browser. Each module and its GUI can be uniquely identified on the network and therefore be executed in any number of instances without creating port conflicts. In this case Aquila GUI simply generates a new module GUI with a unique index. The modules can be distributed in any number of instances on any computer on the network and therefore need to be able to communicate with Aquila GUI regardless of where they run. This is communication is facilitated via YARP network ports, which connect modules’ ports with the ports assigned to their GUIs. The inherent modularity of this cross-platform architecture coupled with its ability to achieve high performance via GPU processors is directly enhancing its scalability, usability and reuse.

The following sub-sections describe the architecture in detail. Each sub-section focuses on different aspect of the architecture such as the structural organisation of project files, grouping of functionalities into libraries, the description of design and implementation of GUIs, modules and the communication links between them.

A. Project Structure

Aquila project is made up of several different types of files, which includes files for CMake configuration (CMakeLists.txt), CPU (*.cpp, *.h) and GPU (*.cu) source code, user interfaces (*.ui), resources (*.qrc), module configurations (*.ini) as well as module examples (*.txt). The root directory of Aquila is identified by AQUILA_ROOT environmental variable and contains the main CMakeLists.txt configuration file. Aquila GUI files are located in four different sub-directories (include, /src, /ui and /res) under the aquila directory. Aquila libraries and modules are grouped in sub-directories under /libraries and /modules directories. Each library directory contains a CMakeLists.txt configuration file and a source code located under /include and /src sub-directories while each module directory has additional /conf and /examples sub-directories. Different libraries and modules might have additional dependencies, which need to be satisfied in order to be included in the project, compiled and installed. Those parts that did not meet these requirements will be simply omitted from the project and will not affect Aquila GUI, modules or libraries other than those directly depending on the missing parts.

B. Modules

Aquila modules are configurable programs that can run in multiple instances across network, be used with or without GUIs and that have the option to execute their code on GPU processors. All Aquila modules have interfaces providing access to their functionalities and facilitating communication between them and other entities. Each module’s implementation of this interface can be found in their Interface class (interface.h and interface.cpp) inheriting from QThread to enable communication via Qt signals/slots as well as listening on the input ports. In addition to the interface, each module would typically have another class ModuleName (ModuleName.h and moduleName.cpp) inheriting from QThread to enable execution of module’s function in a separate thread and remaining consistent with Qt signal/slot communication system. This class provides the actual functionalities via heterogeneous CPU-GPU code. In the case that these functionalities are already present in one of the Aquila’s libraries (see section III-D) and no additional functions are required, then
Aquila module can leave out the ModuleName class altogether and directly call the library from the Interface.

Modules are configured from their config.ini file located in /conf sub-directory. These files would typically contain default values for various parameters (e.g., robot iCub/iCub-Sim), which can be overridden from terminal or later via network ports. Each module has at least one input and one output port through which it communicates with its GUI and/or other modules (see section III-E). Regardless of specific functionalities, each module needs to be able to return a list of GPU devices found on the system where it is running and change its execution mode (between CPU, GPU and multi-GPU) if applicable.

C. Graphical User Interface

Aquila is an application that provides management tools, communication and GUIs for Aquila modules (see section IV). Aquila comes with a GUI, which is inherently dynamic and was designed to be easy to use, intuitive and clean with all the complexities hidden unless required. Aquila provides a default GUI regardless of whether any Aquila modules are available or not. This default GUI allows users to start different tools, probe local and remote servers, see their parameters and detect any Aquila modules that could be launched on them. Available modules can be easily added using shortcuts (CTRL+T or CMD+T on OSX), from the main menu (/File/Add tab) or the context menu by right-clicking on the tab bar and selecting Add tab options. Any of these actions will result in opening of a dialog where users can select which module to start and where. Once users select a module from a list and a server where the module executes, Aquila will try to establish a connection with it via YARP ports. If Aquila successfully connects to the module, a new tab with a module GUI is added. From this point on, all of the actions triggered in the module GUI are directly linked with the module and vice versa. See section III-E for detailed information about communication between modules and GUIs.

As already mentioned, any number of modules can be launched in any number of instances and on any available servers (see fig. 2). Each module GUI has its own menu, which is always displayed when the module tab is selected. When a user selects another tab, a new menu will replace the current menu. However, those menu elements that are essential for Aquila will be integrated in module menus. For example, these can be options to exit Aquila (File/Quit), view servers (View/Servers), run tools (Tools) or see help in (Help/About). All the menus make a good use of shortcuts, which saves a lot of time when some actions need to be done fast or repeatedly.

Aquila GUI starts by initiating local server and probing all other available servers on the network. These servers are based on a cross-platform implementation of YARP, which allows modules to be executed on remote computers running Linux, OSX or Windows. In addition, Aquila collects information about computers running servers such as their specification, current utilisation and available Aquila modules. No modules are automatically added by default, however, this can be changed by starting Aquila with specific arguments. By default, Aquila logs all its activities into a log.txt file. These are typically messages printed by Aquila functions, servers and
modules. No messages are printed to terminal unless specified by arguments.

Fig. 2. Aquila running multiple modules locally and on two different servers. Notice that those tabs without any numbers in brackets are those that run locally while those with identification numbers are running on remote servers. The first number represents instance and the second number server identification. Local modules can have only one number in brackets, which would represent their instance identification. However, in this particular case, the four modules are running the very first instances in which case no numbers are shown.

D. Libraries

The common functionalities are grouped into GPU-accelerated libraries located in /libraries directory. Aquila libraries use Qt and inherit directly or indirectly from QOBJECT, which makes it consistent with Aquila and its modules. More importantly, this allows Aquila libraries to emit and receive signals via the already mentioned signal/slot solution, which is very convenient way for interfacing objects.

Libraries that are currently implemented include:

- **libaquila-nnet** - implements neural networks and their training algorithms (e.g. multiple time-scales recurrent neural network, echo state network, self-organising map, backpropagation through time)
- **libaquila-image** - implements functions related to image processing (e.g. saliency detection)
- **libaquila-utility** - implements various different utilities that are commonly used (e.g. utilities for communication, GPU handling, maths)

E. Communication

This section covers two main systems that Aquila uses for communication. The first one is based on YARP and it is used for communication between processes distributed across any number of machines. The second one is based on Qt and it is used for communication between objects.

1) **YARP Ports**: Aquila needs to be able to communicate with its modules and vice versa. Aquila modules and their GUIs have dedicated interfaces providing communication between them via YARP ports. Every interface listens for incoming messages and sends out data when necessary, which is achieved via named entities, called "ports". YARP ensures that if one knows a specific port name, it is all that is necessary to communicate with it. Just as Internet DNS name service converts domain names to IP addresses, YARP name server (YNS) allows communication between ports providing that users know their names. A port can send data to any number of other ports. A port can also receive data from any number of other ports. The communication between ports can use different transports and protocols (TCP, UDP, multicast, shared memory or text mode) and can be freely added or removed.

YARP communication is based on the observer software design pattern in which an object maintains a list of observers and notifies them automatically of any state changes usually by calling their methods. This design pattern is used by many different libraries, frameworks and almost in all GUI toolkits. Following the observer design YARP port objects are capable of delivering messages to any number of other port objects (observers) in any number of processes distributed across any number of machines using any of the available underlying communication protocols.

Aquila implements interfaces for modules and module GUIs where each interface has at least one input and one output port. Module GUIs send data to modules via output ports and receive data from modules via input ports. Similarly, module interfaces receive data from module GUIs via input ports and send out data via output ports. Aquila ensures that the interface ports have always unique names. Module GUI interface names its ports following this pattern /aquila/hostName/moduleName/instanceID, which is then suffixed with :i for input ports or :o for output ports. On the other hand, module interfaces follow this pattern: /moduleName/instanceID suffixed with :i for input ports or :o for output ports. This unique naming ensures that modules can be running in any number of instances without causing port name conflicts.

2) **Qt Signals and Slots**: In GUI programming, when one widget changes, other widgets often need to be notified. More generally, it is necessary that all types of objects are able to communicate with one another. Previously, this type of communication was achieved using callbacks, which are pointers to functions. Callbacks are normally passed to a processing function, which is then able to call the callback when required. Callbacks are strongly coupled with processing functions as they need to know which callbacks to call. In addition, callbacks are not type-safe as it is not possible to be certain that the processing functions will call the callbacks with the right parameters.

In order to address these limitations, Qt developed an alternative solution to callbacks based on signals and slots. A signal is emitted when a particular event occurs, while a slot is a function that is called in response to a particular signal. An object that emits a signal does not know if anything receives it just as a slot does not know if it has any signals connected to it. Signals and slots mechanism is not only loosely coupled but also completely type-safe in contrast to callbacks. This is because the signature of a signal must match the signature of a slot allowing compiler to detect possible type mismatches.

Signals and slot mechanism has been adopted by Aquila to communicate between different types of objects. Connecting
Fig. 3. Server monitor displays information about servers where Aquila can spawn its modules. The left part shows the standard system information retrieved via yarp run –sysinfo command. The middle part shows the properties of detected NVIDIA GPU devices that can be used by modules. The right part shows current CPU and memory usage.

all GUI widgets via signals and slots is the classical way of using this mechanism. In addition to this, Aquila and its modules use this system extensively to communicate with their Interface objects that are sending and receiving messages over YARP network. When an interface receives a new message, a specific signal is emitted and required slot called. Different signals are aimed for different objects. For example, messages related to module settings would typically emit a signal directly passing received values to a slot defined in the settings object’s class.

IV. MODULES

This section briefly describes modules that are currently present in Aquila and implement their corresponding GUIs.

A. ERA - Epigenetic Robotic Architecture

Epigenetic Robotics Architecture (ERA) is a hybrid cognitive architecture dynamically generating spreading activation models (IA and IAC) not dissimilar to those hard wired in early Connectionism. ERA models continuously learn attempting to predict multimodal and sensorimotor contingencies. ERA makes use of the SOM and ESN modules within Aquila and dynamically grows as new streams of input arrive at its incoming YARP port.

B. MTRNN - Multiple Time-scales Recurrent Neural Network

Multiple timescales recurrent neural network (MTRNN) model attempts to overcome the generalisation-segmentation problem through the realisation of functional hierarchy that is neither based on the separate modules nor on structural hierarchy, but rather on multiple timescales of neural activities that seem to be responsible for the process of motor skills acquisition and adaptation as well as perceptual auditory differences between formant transition and syllable level. Aquila implements a multi-GPU version of MTRNN together with a back-propagation through time training algorithm.

C. Tracker

Tracker provides an image differencing engine and a reliable motion tracking system. Addional settings allow dynamic adjustment of thresholding function, which modifies the level of sensitivity to changes in the visual field.

D. SOM - Self-organising Map

Self-organising map (SOM) is a type of artificial neural network trained using unsupervised learning algorithm. SOM is able to produce a low-dimensional discretised representation of the input of the training samples. Self-organising maps can provide low-dimensional views (2D, 3D) of high-dimensional data, which is particularly useful for visualisations.

E. ESN - Echo State Network

Echo State Network (ESN) is a recurrent neural network hidden layer of which is sparsely connected usually with around 20% connectivity. The connectivity and synaptic weights of hidden neurons are randomly assigned and are permanent while the synaptic weights of output neurons are able to learn to (re)produce specific temporal patterns. This module is a simple implementation of ESN complete with bi-directional readout feedback and input training to provide a reversible system.

V. CONCLUSION

The modelling of the integration of various cognitive skills and modalities requires complex and computationally intensive algorithms running in parallel while controlling high-performance systems. The distribution of processing across many computers has certainly advanced our software ecosystem and opened up research to new possibilities, which was essential move. In order to augment the possibilities within cognitive robotics research, we have developed Aquila 2.0, an open-source, cross-platform software architecture that makes
use of independent heterogeneous CPU-GPU modules that can run anywhere across the network in any number of instances and using any number of available GPU devices. We have described the main features and contributions of Aquila 2.0 rooted in its fundamental design giving rise to hi-performance, scalable, modular yet easy-to-use and develop for platform ambitiously aiming to enhance the field of cognitive robotics research.

Our future plans are equally ambitious. We will continue with the refinement of the architecture, adding new features desired by our research team, users and developers. New large-scale cognitive robotics models are currently being development by our team. Once they are ready and tested, more modules and libraries will be added to the project. We anticipate that over the time, as more of these, hi-performance, scalable modules emerge; we will develop Aquila meta-language providing a hi-level of abstraction allowing the emergence of novel models that were previously unthought of.

ACKNOWLEDGMENT

We would like to thank to EU FP7 Italk and Poeticon++ projects for supporting this work.

REFERENCES