A Framework for Design and Implementation of Control Software

Jorge Estrela da Silva
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida - 431, 4200-072 Porto
Portugal
jes@isep.ipp.pt

Abstract

This paper describes a framework which was used in the implementation of software for the control of autonomous vehicles and whose principles could make it suitable for other kind of applications. The framework acts as a layer over the operating system, hiding its relative complexity from the developer. The framework is based on a location transparent multicast message passing paradigm. This mechanism is valid for communications between modules residing on the same processor or on different machines, making it also a candidate for distributed systems. The current version of the framework is implemented over the QNX 4.25 real time operating system.

Key Words
Software Methodologies, Applications, Software for Distributed Systems.

1 Introduction

Nowadays, control systems are mainly built using the computer based approach. The computer system is interfaced to a set of sensors and actuators. Additionally, the control system is completed by the control software that runs on the computer or even on a microcontroller, for simpler applications. This architecture implements the basic sense-calculate-actuate control cycle.
The development and analysis of new control algorithms and strategies is an active topic of Research and Development. However, increasing the complexity and robustness of the control system, while maintaining its computational efficiency and dependency, has dramatic effects in the software development costs. For that reason, software engineering, computer science and commercial development have started directing some efforts towards the creation of automated tools for the design and implementation of this type of applications.
Generally, developing software applications for the control of multi-input multi-output systems is not a trivial task. In practice, we may find a plethora of physical devices (sensors and actuators) with different interfaces, sampling rates and communication protocols. This multitude of devices, more often than not acting asynchronously, generates concurrency problems, since at least some of them will be centralized in the same computer. Additionally, we may have to perform synchronization and data exchange between subsystems running on the same or different computers.
Even when the control system is to be contained in the same physical system (for instance, on a plane), it may become distributed due to processing and packaging requirements. In the limit, the control system may change in dimension and "span", for instance as in an air traffic management control or multiple vehicle formations.
Most control applications are structured as a set of components obeying to some logical hierarchy or interconnections (for a survey on control architectures see [1]). However, most control engineers, and even software engineers, are not prepared to face the task of transforming a block diagram in a software architecture and to proceed to the implementation of a software application. Even when an initial approach has some success, normally problems will arise as application has to scale and meet programming styles of different developers. For a more thorough discussion about software architectures and possible approaches see [2] and [3].
One possible approach to cope with these difficulties is to develop control oriented software framework. The idea behind these frameworks is to enforce an architectural style and to hide the computational issues (scheduling of processes, priorities, interprocess communication, etc.) from the control engineer implementing the software.

This paper presents a software framework whose main characteristic is its simplicity, from the point of view of final users. Much effort has been applied in trying to create generic APIs that will fit any kind of application or development style, or that simply try to find a common nomenclature for the primitives of the typical operating systems. The approach described on this paper goes on the opposite direction and tries to impose an architectural style which we believe to be generic enough to build many types of control systems.
The framework consists of a set of core modules and of a static library to be linked to the application processes designed and implemented by the author. The services

provided by the library act as a layer between the application and the operative system. It was implemented and used over the QNX 4.25 real time operating system.
The structure of the paper is as follows. Related work is presented in section 2. The framework is described in section 3. Section 4 illustrates the usage of the framework in an application for control of underwater unmanned vehicles and section 6 finally presents the conclusions.

2 Related Work

2.1 Time-Triggered versus Event-Triggered Architectures

Traditional basic theory for computer controlled systems assumes an ideal scenario, with a single sampling rate for the control system and that new samples are always ready at each cycle in order to calculate the control action. Multi-rate system theory allows a discrimination of faster and slower modes in order to apply only the necessary sampling rate on each case. More recently, there is a trend [4] towards analysis of event driven control systems. In this approach, control actions are only taken after some significant change of the environment or system state.
In what control software is concerned, the two above mentioned control theory trends may be loosely related with the contest between event-driven and time-driven approaches [5]. Time triggered approaches are more attractive from a determinism and analysis the point of view. There is a whole sound theory for scheduling of periodic processes [6]. For aperiodic processes, the scheduling analysis becomes more difficult and may require some oversimplification of the problem. A usual approach is to convert aperiodic processes to periodic processes, assuming the worst case execution rate.
The main problem of the time triggered architecture is that we have to build systems rated for worst case execution rates that may be only really performing necessary processing on a little percentage of the total working time.
When considering systems with energy constraints, such as autonomous vehicles, we defend that the event-driven approach allows a more adequate usage of resources. Obviously, the worst case scenario must be considered. However, an event-driven system may provide a higher energy efficiency if allied to special hardware (e.g., delta-sigma converters for analog signals, event driven instruments, CPUs with advanced power management) by running the software only when necessary. In the worst case an event driven system can behave like a time-driven system (where the main event are clock ticks). This enables an evolution of the system pair to pair with the technological and theoretical development.

2.2 Development Platforms

Ideally, the control engineer would like the platform to behave like a continuous equation or, at least, like the ideal sense-calculate-actuate cycle described above. Most of the control designs are made in continuous time. Asking to discretize the equations sometimes is already regarded as “asking too much”.
An approach that is frequently defended is to let the control engineer design the control system in the continuous domain (or even “hybrid” if some type of switching decisions are necessary) and to resort to automated software generation toolboxes for code generation. Commercial tools like *Real Time Workshop* from *Mathworks* offer some support in this area.
An even greater flexibility in designing control applications is proposed, for instance, in the *Ptolemy* approach [7]. *Ptolemy* is a software development tool targeted for control applications, offering several domains for modeling different components of the application. This tool follows a software based components approach, allowing the transparent interconnection of heterogeneous components.
On a lower level of abstraction we have methodologies, such as the PBO model [8], that simply enforce a common structure for the application’s components. The main objective of this methodology is to enhance reutilization of the software components and to provide a framework for dynamically reconfigurable software. Each component must implement a set of methods which are used by a custom scheduler to control the execution of the application. The PBO model was initially developed to run over the research real time operating system Chimera. The RCS library [9], with a greater support for distributed systems and a basic graphical interface for application deployment support, is another approach of this kind. Another example, with a greater concern and support on formal analysis is the Orccad system [10]. The two latter tools assume that some supported operating system will be used.
The OSEK/VXD specification for Operating Systems and Communications [11] was created and proposed as a standard for the development of software for distributed control in the automotive industry and is available in the form of commercial products. This platform can be used in a heterogeneous range of environments which may go from 8 bits embedded microcontrollers to desktop processors.
The main problem with most of these products, mainly the ones from academic research, is that they are usually well fit to a certain platform or configuration, used to prove the concept, but lack a general support for the extreme diversity of hardware products used in the industry. The difficulty in tailoring existing tools to the specific needs of our applications was the main motivation for this work.

3 The Framework

3.1 Architecture

The framework proposed on this paper consists of a set of core modules (*EventManager, Config and xxx_comm* modules) and of a library which is linked with the application's processes. The framework and respective applications are meant to be executed over an operating system, preferably a multi-tasking one.

The framework is inspired on the paradigm of discrete event systems. The elementary processing entities of this framework are called modules. Each module can be seen as having an execution thread of its own, which can be represented by a state machine. A control application will consist of one ore more modules. These modules may synchronize and communicate between them using the event propagation mechanism.

In this context we define *event* as special kind of message that is composed by a numerical identifier, a timestamp (indicating the instant of generation of the event) and optional variable size data.

The event propagation mechanism consists, from the developer point of view, only of two primitives: *generate_event* and *make_event_dependent*. For efficiency purposes, each module must define the events it should react to, using the primitive *make_event_dependent*. The events are generated using the primitive *generate_event*. Although with a very different implementation, this event mechanism is somewhat inspired by the event model of the Teja tool [12].

Any module may generate or be notified for events but, in principle, the events will be generated by modules sampling the environment, by modules making calculations based on values sampled over time or by mere time-driven modules. Each event is identified by a numerical tag which is defined at source code level.

The event mechanism is asynchronous. In a distributed system, we assume that clocks are synchronized or that any procedure that is sensitive to causal ordering will be handled at application level.

The modules are classes which refine the base class *Module*. Each module class must implement the *initialize* and *process* methods. Additionally, the constructor must accept a defined set of arguments in order to enable retrieval of parameters from the configuration file, as explained below.

The configuration file is a text file which defines some computational aspects of the application, like priorities and execution periods (for periodic processes) and also application level parameters (thresholds, controllers' parameters, etc.). Each module has its own section, in a format that resembles the Microsoft Windows 3.0 *ini* files. These sections are referenced by the order of appearance in the file and that information must be passed to the module at instantiation time in order to be possible to fetch the respective parameters. The configuration file is managed by the module *Config*. This module manages the concurrent access to the configuration file and allows for the file to be retrieved from other machine (useful, for instance, on a system without writable storage).

There are several ways of packaging the modules of one application:

- In a single-tasking operating system, the modules will be managed in a light-thread fashion, without support for preemption. The developer will have to avoid the usage of blocking functions and should also use a style of programming with short computations in order to allow an as fine grained as possible execution of each module by the framework's kernel. Even in a single-tasking operating system we may have some kind of preemption, namely if we resort to interruption routines. Applications using interruption routines end up behaving like a small operating system or executive kernel and may become difficult to maintain.
- In a multi-tasking operating system without built in support for threads, it is possible to package the critical modules in separated processes and to use a more liberal style of programming. It is still possible to package several modules on a single task or even on several tasks (but without preemption support). This would be done mainly due to efficiency purposes: if there is a set of modules that does not concur for processor utilization, it is advantageous to package them on the same module thus avoiding scheduling and inter-process communications overhead.
- In a multi-tasking operating system with kernel support for threads we have a trade off between the two above mentioned situations. We may allocate critical modules to different threads thus enforcing isolation of execution of each module and, at the same time, avoiding the use of inter-process communication primitives, since data variables reside on the same memory space. Even with this kind of thread support, there are scenarios where it is advantageous to divide the application in multiple processes, namely if we are interested in replacing or adding new parts of the application without shutting it completely down. Most operating systems do not allow to dynamically link and execute a new binary module on an existing process. However, it's easy to kill a process and to launch a new one. That's why it is not always a good option to package the whole application as a single process.

The list of processes and respective modules to be loaded are indicated in the configuration file. The library provides a *stub* for the main function of the process, which may be useful for processes with multiple modules. The developer may bypass the stub and create the main

function by him/herself. This stub loads the modules instances as specified in the configuration file. A detailed description of the syntax of the configuration file (as well of the primitives of the framework) can be found in [13].
After the typical initialization phase, on which the initialize method is invoked (for setting priority, instantiating auxiliary classes, opening ports, etc.), the process consists of a cycle with a call to *wait_event* (which is a blocking call). Whenever the process receives a new event (after some *make_event_dependent* configuration), the *wait_event* call returns. Based on the event identifier, the module(s) which registered for that event have their *process* method invoked.

3.2 Implementation of the Event Mechanism

The current implementation of this framework is targeted for the QNX 4.25 real time operating system. This is a multi-process operating system which follows the traditional UNIX programming model (most of kernel calls are POSIX 1003.1 compliant) with additional real-time extensions (POSIX 1003.1b, former POSIX 1003.4). Anyone used to programming UNIX applications [14] will find the development environment familiar.
The framework is implemented in C++ and uses the real-time primitives of QNX. This way, we assure that priority policies, regarding scheduling and communication between tasks, are assured.

The event mechanism is managed by the framework module *EventManager*, which must run on each processor. In a multi-process operating system, this module manages a dynamic list of events that, when generated, must be propagated to other processes in a multicast fashion. Each entry in the event list points to a list of dependent processes. Whenever a module executes a call to *make_event_dependent*, a new entry is created on this process dependency list (PDL) for the respective event, if it does not already exist, and the identifier of the process containing the module is added to the process list.
The above mechanism is not very efficient if a process has several modules, since each module will receive the same event and thus will have to be able to filter the incoming events. A new version of this framework is being developed in which each process may contain a local *PEventManager* module. The *PEventManager* holds a PDL like the one described for *EventManager*, with the exception that local dependencies are not stored. These dependencies are stored in a different list, the local module dependency list (LMDL) where references are made to module pointers instead of process identifiers. The *PEventManager* module registers itself in the central *EventManager* at process initialization. When a module executes the *make_event_dependent* call, the LMDL is updated, if the *PEventManager* is used on that process. Additionally, the *EventManager* performs a *make_event_dependent* call to each *PEventManager* of the processor thus replicating the PDL on each *PEventManager*. When a module executes a *generate_event* call, two situations may occur: if a *PEventManager* is not present in the hosting process, the call will behave as before; if a *PEventManager* is present, the procedure will check the LMDL, in order to serve the local modules (with no context switching), and then will check the local PDL in order to propagate the event to other processes, without any access to *EventManager*.

Communication between different processors, in a loosely connected system, is made by special communication modules. The current implementation supports three modules: *udp_comm*, *rs232_comm* and *can_comm*. The principle of operation is the same for all of them but they can be tailored for the specific characteristics of the respective communication media. For instance, the *udp_comm* is meant for connectionless communication over an ethernet network. These modules act like a bridge between processors having two PDLs (incoming and outcoming).
When a local *make_event_dependent* call is received by the *udp_comm*'s *PEventManager*, the module updates the incoming PDL and propagates the call through the network to a central processor on which resides the module *NEventManager*. This module has a role in the network similar to the *EventManage's* role in the processor. When *udp_comm* receives an incoming event from the network, it uses its incoming PDL, to propagate it to the local processes. When *udp_comm* receives a remote *make_event_dependent* call, it updates the outcoming PDL and makes itself dependent on that event.
At this time some questions could be put about the best allocation of channels since in a control application we can have channels between processors with extremely different capacities (compare for instance the connection between two computers in a vehicle with the link between a satellite and a ground station). At this time, the treatment of that issue will be hard-coded at *NEventManager*. However, this is a topic for future work.
The first version of this framework used a message queue per process allowing buffering of all messages. However, in real time control systems there are many situations where we are only interested in the most recent data. In those situations the buffering offered by the message queues was an unnecessary overhead. The message queues were replaced by a shared memory heap managed by the *EventManager* module, with a "slot" for each application's message. In this alternative version, only the most recent event (of each type) is delivered. The framework maintained its interface and no change (besides rebuilding the processes) had to be made to the control applications. A future work will be to modify the *make_event_dependent* call and underlying library's structures in order to allow the selection between buffered and non-buffered reception.

As can be seen, the framework takes care of the location transparency problem. When an application module generates an event, it does not need to know where the receptors are. Therefore, the developer does not need to choose between different primitives for remote and local broadcasts. Even for communications inside the same machine, no addressing or process identifiers are needed. The only arguments of *generate_event* are the event identifier and the optional message. This implies that a specific module does not need to know anything about the remaining application, namely about the eventual receptors of the event. This mechanism resembles the OSEK/VXD communications model [11] with the difference that, in this paper, message configuration can be done at run-time, throughout the whole life cycle of the application. In OSEK, the message routing configuration is done statically for efficiency purposes.

4 Applications

4.1 Underwater Vehicles

The framework was employed on the development of the control software for the LSTS' Autonomous Underwater Vehicles (AUVs) and Remotely Operated Vehicle (ROV) [15]. A preliminary description of this work can be found in [16]

These are electrically powered vehicles, with a PC104 computer stack which comprises a PC motherboard, a hard disk and, depending on the type of vehicle, a subset of the following boards: serial port boards, analog to digital conversion boards, ethernet boards, CAN boards, HDLC boards, digital to analog conversion boards, etc. These boards are required to interface with the system's sensors (accelerometers, pressure sensors, compasses, sonars, etc.) and actuators (DC Motors, lamps, sound emitters, etc.).

The application consisted of a dozen of modules running on a Pentium processor at 166MHz for the ROV system and on a 486DX4 at 100MHz for the AUV system. Both systems use QNX 4.25. In the AUV system, the performance could be compared against a previous version of the application which ran over DOS. The overhead of the new application plus the multi-tasking operating system was noticeable on the 486 system (the approximate value of 20% is given, since it was difficult to get an accurate measure for the DOS application). In the Pentium system the overhead is less noticeable, in part due to the higher frequency and probably due to a better optimization of the operating system for this processor (that can be inferred, for instance, from the context switching and interrupt latency times indicated on the OS' manuals). The application that runs on the ROV is the same that runs on the AUV with the exception of the module that manages the autonomous execution of missions (which is not employed in the ROV) and some hardware drivers.

It was observed that the framework is easily understood by other members who join the software development "task force" or who simply desire to adapt the control application in order to accommodate their ideas and experiments.

4.2 Example of Distributed System

The LSTS ROV receives commands from an external console. This console consists on a laptop with one joystick running an application which graphically displays the state of the ROV. This application was developed by one of the LSTS team members using the programming language Java and it runs over Microsoft Windows XP, since there are no hard timing constraints in what concerns displaying data. The *generate_event* and *wait_event* primitives were reutilized from an existing application developed in C++ and packaged as a DLL. The console application resorts to the Java Native Interface (JNI) in order to invoke the primitives. Some care had to be taken in order to convert binary data between the vehicle and console platforms. Although both computational platforms use *little endian* multibyte number format, Java always assume *big endian* format. JNI automatically converts between formats but only for single data variables with explicit identification of the data format. This issue was solved by developing a parser routine which converts the binary data structures to an array of fields in the Java side. The parser knows the format of the data structure associated with each event. This meta-information is provided by the definition (.def) file which defines the structure of the events' message. Upon receiving an event, the parser performs a cycle, reading each individual field of the structure and calling a JNI method with the associated data type (for instance, *readDouble*() for double data type field).

This problem arises because, in order to avoid additional processing, data structures are broadcasted in native format instead of using a common network format. We must remember that when dealing with autonomous systems extra processing may have direct impact on system autonomy.

On the other hand, the communication with console was added to the original application without any modification of the existing modules. Only a new module was added, the framework module *udp_comm*. The console simply registers itself for all the interesting events and they are forwarded to it by the *udp_comm* module.

5 Conclusions

The framework presented in this paper hides software structural aspects from developers. This is a desirable property when we are dealing with a team whose main expertise is not software development. The framework enforces a discipline and architectural style, hiding the API of the underlying OS, which in practice has proved to enhance team work.
The event mechanism allows a multicast type of message passing which is normally not found in typical operating systems. The location transparency inherent to the event mechanism greatly simplifies the development of software modules and we still did not find any major problem with this approach considering the target applications. The non expert user is spared from dealing with RPCs, message queues, shared memory, etc.
The adopted architectural style imposes an extra overhead which is, however, an expected price to pay and compensated by the greater flexibility when customizing and reusing applications. This framework also lacks the support of a graphical user interface for system composition and management of components.
The framework was successfully used in the development of the control software for underwater vehicles and we think that at least some of its principles could be useful in the development of other medium or high complexity control applications.

6 Acknowledgments

Part of this work was developed during the elaboration of the author's master thesis, having Prof. Fernando Lobo Pereira as advisor. The author would like to thank João Sousa for many insights in the state of art of this paper's topic. The console application mentioned in the distributed system example was implemented by Sérgio Fraga.

References

[1] J. K. Rosenblatt, *DAMN: A Distributed Architecture for Mobile Navigation* (PhD Thesis, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1997)

[2] Robert Allen and David Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, 1997

[3] Bridget Spitznagel, and David Garlan, A Compositional Formalization of Connector Wrappers, *2003 International Conference on Software Engineering (ICSE'03)*, Portland, Oregon, USA, 2003 3-10

[4] K. J. Aström, B. M. Bernhardsson, Comparison of Riemann and Lebesgue Sampling for First Order Stochastic Systems, *Proceedings of the 41st IEEE Conference on Decision and Control*, Las Vegas, Nevada, USA, 2002

[5] H. Kopetz, Should Responsive Systems be Event-Triggered or Time-Triggered, *IEICE Transactions on Information and Systems*, *76*(11):, 1993, 1325-32

[6] Ramamritham, John A. Stankovic, Scheduling algorithms and Operating Systems Support for Real-Time Systems, *Proceedings of the IEEE*, 82(1), 1994

[7] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, Yuhong Xiong, Taming Heterogeneity - the Ptolemy Approach, *Proceedings of the IEEE*, 2003

[8] D. Stewart, R. Volpe, and P. Khosla, Design of Dynamically Reconfigurable Real-Time Software Using Port-based Objects, *IEEE Trans. Software Engineering*, 23(12), 1997

[9] V. Gazi, M. L. Moore, and K. M. Passino, `Real-time control system software for intelligent system development: Experiments and an educational program, *IEEE Int. Symp. on Intelligent Control*, Gaithersburg, MD, 1998, 102-107

[10] D. Simon, B. Espiau, K. Kapellos, R. Pissard-Gibollet, Orccad: Software Engineering for Real-time Robotics, a Technical Insight, *Robotica - Special issues on Languages and Software in Robotics*, 15(1), 1997, 111-116

[11] *OSEK/VDX Communication, Version 3.0.1* (OSEK Group, 2003)

[12] A. Deshpande, The DIADEM System for Real-Time Dynamic Event Management, *Proceedings of NATO Workshop on Discrete Event and Hybrid Systems*, Antalya, Turkey, 1997.

[13] Jorge Estrela da Silva, *Software para Controlo de Veículos Autónomos*, (Porto: Porto University - Msc Thesis, 2002)

[14] Bill O. Gallmeister, *Programming for the Real World – POSIX.4* (O'Reilly & Associates, Inc., 1995)

[15] N. Cruz, A. Matos, J. Borges de. Sousa, F. Lobo Pereira, J. Silva, E. P. Silva, J. Coimbra, and E. B. Dias, Operations with Multiple Autonomous Underwater Vehicles: the PISCIS Project, *Proceedings of the Second Annual Symposium on Autonomous Intelligent Networks and Systems*, Palo Alto, USA, 2003

[16] J. E. Silva, A. O. Martins, F. L. Pereira, A Reconfigurable Mission Control System for Underwater Vehicles, *Proceedings of the MTS/IEEE Oceans'99 Conference*, Seatle, WA, USA, 1999.