Herein is proposed a complete software architecture for autonomous vehicles, from the development of a high-level multiple-vehicle graphical console, the implementation of the vehicles’ low-level critical software, the integration of the necessary software to create the vehicles’ operating system, the configuration and building of the vehicles’ operating system kernel, to the implementation of device drivers at the kernel-level, specifically a complete Controller Area Network subsystem for the Linux kernel featuring a socket interface, protocols stack and several device drivers.

This software architecture was fully implemented in practice for the Delfim and DelfimX autonomous catamarans developed at the Dynamical Systems and Ocean Robotics laboratory of the Institute for Systems and Robotics at Instituto Superior Técnico. The DelfimX implementation is described and real profiling data of the vehicle’s software performance at sea is presented, showing actuation response times under 100 microseconds for 99% of the time and 1 millisecond worst case with 10 parts-per-million accuracy, using a standard Linux kernel.

Keywords: software, autonomous, Linux, DelfimX, CAN.

Introduction

Through recent history, humankind has been designing autonomous vehicles to perform tasks that are too monotonous, meticulous, dangerous, or simply not possible for a human to execute, and do them without the need of constant human supervision. From the deep seas, with the SPURV submarine as early as 1957, to the outer space, with the Mars rover Opportunity in 2004. Initially, autonomous vehicles were almost entirely hardware-based, which made the development process slow and expensive, and the final product restricted to its initial goals, not being easily upgradable or extendable to perform even so slight different tasks. With the advent of software, autonomous vehicles became increasingly powerful. Most now use common off-the-shelf hardware. The software is nowadays the brain of any autonomous vehicle.

When building an autonomous vehicle, a significant part of the investment therefore goes to the software development process. To make it the most profitable, a proper software architecture for autonomous vehicles is greatly desired, one that already solves the common problems faced when implementing or integrating the software for autonomous vehicles, otherwise it becomes a colossal task. It is not just writing code that implements a particular algorithm that makes the vehicle do this or that. It is building, integrating, writing, simulating, testing all the necessary software components that make the vehicle a workable platform in the first place.

The diagram in Figure 1 depicts the central role of software inside an autonomous vehicle, with its supporting hardware components that are needed to interface with the environment, and is valid for near all autonomous vehicles.

![Fig. 1 The role of software in an autonomous vehicle.](image)

Hardware sensors are needed to perceive the environment (these are inputs to the software) and hardware actuators are needed to perform in the environment (these are outputs of the software). Optionally, communications hardware can be used to interact with the vehicle in real-time (these are both inputs and outputs for the software). In practice, some form of communication link is usually always present in any autonomous vehicle. Optionally as well, but very common and sometimes even fundamental, is hardware to store onboard log data of the vehicles’ activities and findings (these are outputs of the software).

The problem that is addressed here is exactly how to develop this vital component of an autonomous vehicle that is the software.

State of the art

The art of software programming is very underestimated by the scientific community. As a consequence, when designing an autonomous vehicle, the software implementation phase is often neglected until the last minute, usually leading to questionable ad hoc solutions, including poor choice of operating system, development en-
environmens that are not suitable for autonomous vehicles, over-complex implementations using objects, threads, synchronisation, remote method calls, state machines, all of which are best avoidable in the first place, to the utmost absurd of running complete software simulation environments onboard real vehicles. Some even use proprietary software, which is incompatible with a sustainable evolution of civilisation.

Much work has been done on individual software units for autonomous vehicles, like inertial measurement units, navigation units, control units, or specific algorithms, such as obstacle avoidance, target tracking, environment mapping, but not on how to actually integrate all this software into a working autonomous vehicle.

The small part of the problem that is sometimes addresses is how to implement only the main program that controls an autonomous vehicle. Figure 2 shows the typical implementation.

Fig. 2 Typical implementation.

The first controversial issue is the use of threads. For a system to run the fastest, it usually needs to be implemented as a single program running as a single process. For a system to be the safest, it usually needs to be implemented as separate programs running as separate processes. Threads, on the other hand, are neither the fastest nor safest. There is a common misconception that dividing a program into multiple threads will somehow make it run faster. But the reality is that, without multiple processors, running multiple threads will only make a system slower, due to the overhead of scheduling, context-switches and synchronization between threads that the operating system must perform to safely time-share the processor between the multiple threads.

Another common misconception is that programming using threads is easiest, but in fact, the most frequent cause of errors in software is the incorrect handling of concurrency, and when using threads, one has to deal with it himself, correctly, or else there will be race conditions and deadlocks.

While in theory concurrency should be well understood by every programmer, it is many times implemented wrong.

Then, depending on the specific vehicle and its applications, any number of threads would be necessary and they need to somehow interface with each other. In this simple example, with only three actuators and three sensors, the diagram is already rather complex, and in reality, there will probably be more actuators and a lot more sensors. Furthermore, all threads are contending for the same shared memory containing the vehicle’s state data. It will most likely happen that low priority threads will occasionally prevent high priority threads from accessing the data. A typical cover-up improvement is to implement separate shared memories, with their own locks, associated to each producer thread. This way, there is fewer data contention for each data that could prevent a high priority thread from executing, say, a time-critical function. But one still need locks, and the general rule is that if a critical execution path contains locks, it is doomed to perform badly.

Proposed architecture

The software architecture for autonomous vehicles here proposed addresses all layers of the development process, from the implementation of a high-level graphical user interface console for the vehicles, the implementation of the low-level critical program that controls the vehicle, the integration of the necessary programs to create an operating system for autonomous vehicles, optimizing the boot process, the configuration and building of the vehicle’s operating system kernel, to the implementation of vehicle-specific device drivers at the kernel-level.

The first step in the development process is to establish the software requirements of the autonomous vehicles, and the primordial ones are interface support and platform independency. Hardware sensors and actuators exist with a huge variety of interfaces (Ethernet, USB, RS232, I²C, SPI, CAN), and similarly for the communications hardware and data logging hardware. The choice of hardware and software platform should support all required interfaces by the vehicle. Furthermore, the software platform should be independent of the hardware platform, as it should be possible to exchange computer hardware and still use the same existing software without having to write any more code. These requirements are met by opting for a modern, free, operating system kernel, like Linux or FreeBSD, which has drivers for all the above interfaces and runs on multiple platforms.

The boot process, from when the computer is first powered on until it actually starts executing the operating system, is a complex platform-
specific process that is usually neglected and unoptimized, as it was designed for desktop computers, not for autonomous vehicles. The traditional boot process executes the operating system from disk. There are several shortcomings in this approach, so it is proposed a boot process, supported by modern operating system kernels like Linux, that first loads the kernel (≈ 1MB) and file system (≈ 300kB) image files to memory and then runs the system entirely from memory. The main advantages are safety and robustness, as disk operations are inherently unsafe, in particular a power failure while performing a write operation may damage sectors or even render the device unusable, thereby bricking the autonomous vehicle. It also allows new software to be tested onboard and online, in memory only, without destroying the original system of the vehicle, which is safely kept on a (unmounted) disk. And updating the whole software of a vehicle is only uploading two files to the vehicle, which can swiftly be done remotely.

The kernel is the central component of an operating system and is responsible for managing the system’s physical resources shared by the user programs (applications). Autonomous vehicles are often associated with real-time kernels, whose schedulers favour minimal latency in handling prioritized events, instead of overall throughput performance, in order to meet deadline requirements individually. But for most autonomous vehicles, a real-time kernel is really not necessary, as shall be shown ahead. Modern general purpose kernels typically provide platform portability, device support, performance, security, stability, POSIX compliancy, configurability and, most important, ease of use, unmatched by any real-time kernel, and there are several free general purpose kernels available to choose from (Linux, FreeBSD, NetBSD, OpenBSD, DragonFly, OpenSolaris, Minix). Linux was chosen here for the simple reason of the author’s 10+ years experience with it. Out of the box, the Linux kernel is perfectly adequate for an autonomous vehicle, but it can be configured and compiled with options that make it behave more like a real-time kernel, if one wishes so. In multi-tasking kernels, each process has a scheduling priority, a niceness value typically ranging from -20 (most favourable scheduling) to 19 (least favourable scheduling), the default being 0. This simple scheme alone is usually all that is necessary for an autonomous vehicle: one assigns higher priorities to more interactive processes, like sensors and actuators, and lower priorities to the other processes, like data logging and communications.

The root file system contains all the files and programs run in user space by the kernel. Everything an autonomous vehicle needs is a dozen standard utility programs for configuring the system, a couple of daemon programs for providing essential network services, and the few vehicle-specific programs for executing the vehicle-specific tasks, which all easily fit under 500kB. After the kernel mounts the root file system, it executes the first user process 1 called init. For an autonomous vehicle, this should be just a simple script that merely initializes the necessary interfaces and programs. In all, there will be a total of 20 to 30 program binaries on the root file system. So few binaries do not justify the use of dynamic shared libraries. The GNU C library libc.so shared object alone is about 1.5MB. Instead, if all program binaries are statically compiled with dietlibc they all sum to a third of that. But the main advantage is that it becomes easier to maintain, by completely eliminating the infamous “shared library dependency hell”. Least, the execution of static binaries is also slightly faster. The files are then placed on the root file system of the autonomous vehicle according to the Filesystem Hierarchy Standard.

There are just two essential network services required in an autonomous vehicle: a File Transfer Protocol (FTP) server, for uploading new programs and downloading data, and a Remote Shell (RSH) server, to remotely execute programs on the vehicle. These alone make of the vehicle a simple and powerful test platform.

After having set up solid platform ready for test and production of an autonomous vehicle, only now is time to implement the program that controls the autonomous vehicle. Instead of the typical implementation shown in Figure 2, it is proposed a much simpler one, illustrated in Figure 3.

![Fig. 3 Proposed implementation.](image)

This is admittedly trivial. The egg of C: an autonomous vehicle is that, now, one implements Figure 3 directly, instead of creating complex architectures with multiple threads, shared memories, synchronization primitives, to implement something so simple. It is just a single process whose Specification and Description Language (SDL) graphical representation is presented in Figure 4a and Figure 4b shows the corresponding implementation in C language. It is remarkable how simple it is to implement the main part of the critical software of an autonomous vehicle when this architecture is followed.
int main()
{
    struct pollfd pfd[N];
    init(&pfd);
    for (;;) {
        poll(pfd, N, -1);
        if (pfd[0].revents)
            console();
        if (pfd[1].revents)
            sensor1();
        if (pfd[2].revents)
            sensorN();
        control();
        actuator1();
        actuatorM();
        console();
        log();
    }
}

Fig. 4b C implementation.

To implement the software to interface the various sensors, whatever the sensors’ physical interfaces (USB, RS232, ethernet, CAN), in the Unix world they are all equally handled in user-space software with the simple read and write system calls. The only precaution is that, if the sensor is to take part in the critical control path, reading and handling the sensor’s data must be fast to execute, which is assumed true for Figure 4b. However, three-dimensional localization sensors (video cameras, sonars, ladars) are now commonly used on autonomous vehicles, and these deliver data at such high rates that reading and processing it directly from the main loop could increase the latency jitter of the reading of other critical sensors’ data in a way that could exceed the specifications. It happens that the objective of installing such high rate sensors on an autonomous vehicle is usually for offline processing only, and a solution for handling these often called payload sensors is to do it in a separate program, running as a separate process, with a lower priority, independently from the vehicle’s critical program, as shown in Figure 5.

Fig. 5 Adding sensors with special requirements.

Today’s computers and ever clever algorithms make it also possible to process such high data rate sensors online and use its results in the critical control path of an autonomous vehicle. For these cases, a solution, also shown in Figure 5, is to implement the high data rate sensor reading and processing algorithms as a separate program, running as a separate process, and send only the end results to the critical program via some inter-process communication method, preferably a PF_UNIX socket, for its simplicity and speed. This way, the critical program merely sees the end results of the high data rate heavy processing sensor as one more input, which then only takes a few microseconds to read.

Data logging is often an important characteristic of an autonomous vehicle, but writing files is slow, so it is a potentially blocking operation which could crash a vehicle. Unix-type operating systems provide several methods for doing data logging. Their execution times were profiled and a library, liblog, was implemented with the best ones for autonomous vehicles: direct logging using the write system call, compressed logging using zlib’s deflate algorithm, and low-latency compressed logging through a pipe to a low priority child process. The library also takes care of organizing the data by sensor and by time, and makes sure files do not grow too big or too old, automatically closing and creating new ones when appro-
appropriate, so that one can just as easily fetch only the data of a particular sensor for a particular time interval, or a whole day’s data.

It is usually wanted to maintain communications with an autonomous vehicles, for a number of reasons: to monitor the vehicles’ status, telemetry, to visualize the sensors’ data in real-time, to modify the missions on the fly, to remotely stop the vehicle in case of emergency, to upload new versions of programs, to retrieve data log files. But the physical link is generally wireless, which is slow, and therefore easy to saturate and also block the critical control path. Like data logging, communications should be performed only after the execution of the critical control path, as shown in Figures 4a and 4b. The Internet Protocol (IP) it typically used, and in this case, applications that transfer time-critical data should set the IP Type Of Service (TOS) flag to low delay, using `setsockopt` with `IP_TOS LOWDELAY`, and the kernel will make sure those packets will always be transmitted first.

Profiling is a technique for measuring the performance of a program as it executes in the real environment, and currently the most honest way to assert about the performance of the software of an autonomous vehicle. This proposed software architecture makes it very easy to do by simply collecting timestamps, using the `gettimeofday` system call for instance, of the program execution at the points of interest, typically before and after each of the function calls in Figure 4b. At the end of each critical path run, the profiling data is logged together with all the other data. The impact of profiling on the critical program’s execution is thus negligible: less than 1 microsecond per `gettimeofday` system call.

Testing autonomous vehicles in the real world is usually very expensive, in all terms of money, time and effort. Furthermore, a test gone bad with an autonomous vehicle can cause serious damages, people including. Therefore, developing a software simulator for an autonomous vehicle is a true necessity. The objective of a software simulator for an autonomous vehicle is to be able to run as much as possible of the exact same code that is going to run on the real vehicle, but without requiring physical access to the vehicle hardware. This allows conclusive testing of new algorithms for the vehicle with the touch of a button, as many times as needed, in a safe, practical, accessible, comfortable environment, like on any desktop or laptop computer. Figure 6 depicts the process of software simulation of autonomous vehicles. On top is the real software and hardware of the autonomous vehicle. The goal is to replace the hardware components with software and to be able to run both the vehicle computer’s software and the console computer’s software in a single standard computer with no special hardware requirements, which is shown in the middle of the figure and involves simulating the hardware sensors and actuators’ data as well as the physical behaviour of the vehicle, and on the bottom the interaction with the sensors and actuators is still performed on the same program as the real code is run, but the physical behaviour of the vehicle is simulated in a separate modelling program, which communicates with the main simulator using some inter-process communication mechanism, typically an IP socket.

DelfimX

DelfimX is an autonomous catamaran developed at the Dynamical Systems and Ocean Robotics (DSOR) laboratory of the Institute for Systems and Robotics (ISR) in Instituto Superior Técnico (IST), Lisbon, Portugal.

![DelfimX autonomous catamaran.](image)

The requirements for the DelfimX’s software are depicted in Figure 8. Shown are the critical sensors, and for the DelfimX these were all designed to have Controller Area Network (CAN) interfaces. The main actuators were also designed...
to have CAN interfaces. The primary communication link for DelfimX is a long range radio that has an RS232 interface. At the software level, it was decided to use the Serial Line Internet Protocol (SLIP) encapsulation for data multiplexing. Other communication links should also be supported for payload usage, typically WiFi for close range, higher throughput, remote applications and ethernet for onboard guest applications. Data logging is performed on a CompactFlash memory card with an IDE interface.

![Diagram of DelfimX software requirements](image1)

**Fig. 8** DelfimX software requirements.

The DelfimX computer system is a PC/104 with the following specifications: AMD Geode LX800 500MHz CPU, 512MB DDR RAM, CompactFlash, Ethernet, RS232/422/485, and TS-CAN1 interface board with Philips SJA1000 CAN controller.

![DelfimX PC/104 computer](image2)

**Fig. 9** DelfimX PC/104 computer.

The operational requirements for the DelfimX software is that it must handle navigation data at a 5Hz rate, and for each sample, execute the control algorithm and send the calculated actuation to the propellers; this defines the critical execution path, and must complete within the 200ms sample interval time. Data logging is performed after each critical execution path, at the same 5Hz rate, but it is not time-critical. At 1Hz, commands are received from a ground console, which expects the DelfimX software to send back the vehicle state within the next second. All other sensors send their status at a rate of 1Hz or lower, and the DelfimX software need only check that the sensors are good, and if not, warn the ground console and stop the vehicle.

The DelfimX operating system kernel is Linux 2.6.30. It supports all the interfaces specified in the requirements, including CAN, out of the box, except that most CAN devices used in DelfimX were developed at the DSOR laboratory and use special CAN protocols, and for these, the CAN implementation presented in the next sections was added to the otherwise plain Linux kernel of DelfimX, which was configured with the most conservative (high throughput, high latency) settings: `PREEMPT_NONE, HZ_100, N0_HZ`. The latency was measured for DelfimX during real operation to be 60µs on average and worst case below 3ms.

The DelfimX root file system is stored in a gzip-compressed cpio-formatted file, currently less than 250kB in size, that is loaded to memory by the boot loader and ultimately uncompressed and mounted as the root file system by the kernel, as explained previously. It contains the fundamental system programs (`sh, mount, ifconfig, ...`), the `/init` script that configures the system and starts the only 8 programs permanently running on DelfimX: `ftpd` and `rshd`, the essential network service daemons, `httpd`, an HTTP server that provides system statistics, `slipd`, which monitors the radio link, `logd` and `canzd` log all the CAN frames on the bus, `powerd` waits for a CAN frame to shutdown, `cantcpd` is a proxy server that allows remote TCP/IP clients to transmit and receive CAN frames on the DelfimX, and finally, `delfinx` is the critical program, the brains of DelfimX, discussed next. The `delfinx` process is run with niceness 0 (normal priority), while all other processes are run with nicer values (lower priorities). Except for the `sh` Bourne shell and the `e2fsck` ext2 file system checker, all programs in DelfimX were written from scratch by the author of this work.

The DelfimX's critical program `delfinx` is implemented in C as a single process, with the source file structure shown in Figure 10. `navcan.c` implements the reading of the navigation data from a CAN socket and `sysstat.c` the reading of all the other sensors’ data, from other two CAN socket. `prop.c` implements sending the propulsion actuation commands to the starboard and port propellers, through a CAN socket. `log.c` implements the writing of the current DelfimX state to a log file using the `liblog` library. `console.c` implements the reading of the commands sent by the console and the transmission of the current DelfimX state back to the console, through an UDP/IP socket, be it from the radio or ethernet. `grex.c` implements the communication with an
onboard guest computer called GREX, an European project where DelfimX is currently involved. The delfimx program acts as a TCP/IP server that the GREX computer connects to, to receive the DelfimX state and send heading reference commands for delfimx to follow. The control/ directory contains the C source files for all the different control modes of DelfimX, one file for each: heading, go to waypoint1, play mission, GREX, and path following. Finally, main.c implements the delfimx main() function, which glues together all the other C files. This is the critical part of the code, and was implemented following the description in Figures 4a and 4b. The SDL graphical representation of the delfimx main part is show in Figure 11, and the corresponding C implementation is similar to Figure 4b.

Fig. 10 DelfimX source code files.

Fig. 11 SDL of delfimx main.

The critical path starts at the consumption of the Navigation signal, in the SDL, which is triggered by the reception of a sequence of CAN frames sent by the navigation node. This timestamp is obtained from the canzlogd CAN logs as the time instant that the PC/104’s CAN controller generates the hardware interrupt that signals the reception of the last CAN frame in the sequence. The critical path ends after transmitting the CAN frames with the propulsion commands, which is obtained from the delfimx logs as the timestamp collected after sending the Propulsion signal, in the SDL. The difference between these timestamp accurately measures the actuation response time of the whole DelfimX software system. This value is plotted in Figure 12, as an histogram, for 150645 consecutive runs of the delfimx critical path during real operation at sea. Since the navigation sends data at 5Hz, that amounts to a very significative 150645/5/3600 ≈ 8.4 hours of continuous operation on a real environment.

Fig.12 DelfimX actuation response times.

In more than 99.3% of the times, the full critical path takes 100µs or less to execute (the axis scales are logarithmic). There are a few outliers, as expected from any general purpose operating system, however only in less than 0.7% of the times it took more than 1µs, yet less than 1ms, to finish. And only in 1 out of 150645 runs (that is less than 10ppm, or parts per million) it exceeded 1ms. It took 2.674ms, to be precise. If the specifed limit had been 1ms, which would be overkill even for the most demanding vehicles, like helicopters, this program still provided a better than 10ppm accuracy. This confirms that the DelfimX requirements are easily met by the software architecture herein proposed, and that it could easily handle a vehicle with requirements one thousand times more demanding, literally.

Two software simulatos were developed for DelfimX, as shown in Figure 13. On top is the real software and hardware of DelfimX. The complete operational setup is made up of three main computers, each with its own specific hardware: hardware for communications between the console computer and the DelfimX computer, hardware for communications between the GREX computer and the DelfimX computer, and hardware for interfacing the DelfimX sensors and actuators. In the middle of the figure is the DelfimX simulator with a built-in model, where the communications...
between the three computers are replaced with local software connections, making it possible to run the complete DelfimX operational setup on a single computer at any desk. The built-in model simulates the physical behaviour of the vehicle, the sensors’ data and the actuators’ response. On the bottom is the DelfimX simulator with an external model. In this case, the simulator replaces the sensors and actuators’ interfaces with a local software connection to a separate modelling program, which then simulates the physical behaviour of the vehicle.

**Console**

The user interface to an autonomous vehicle is historically called the console. It provides to a human operator the power to easily program complex missions, offline, and then remotely monitor the vehicle during real operation, interact with the onboard sensors and actuators, change mission parameters, or even the entire mission, on the fly, to adapt to the different facing scenarios.

The console is part hardware, part software. The hardware part should be kept to a minimum, use off-the-shelf components, and simply let the software do the real work. Figure 14 shows a picture of the console hardware for both Delfim and DelfimX autonomous catamarans. It is only a small laptop, a radio connected through USB and an antenna. The radio communicates to both catamarans in a point-to-multipoint mode, allowing the console to monitor and control both catamarans, and more, simultaneously, for cooperative multi-vehicle missions.

But what really makes the console happen is the software part. The console application was implemented in C++ and uses the Qt framework library. It resembles very much an ordinary graphical application present in everyone’s desktop computer, and is very user-friendly: the central area of the application shows the mission (the map, the waypoints, the vehicles’ past, present and future locations, obstacles) and on the left side is the vehicles’ minimum indispensable telemetry data, in particular the propellers’ speed graphs (the propellers are the components that usually fail the most); there are no title bars, menu bars, or tool bars that consume precious screen space; all actions, including loading, saving and full mission editing, are performed simply by clicking on the map or desired objects and selecting the appropriate items in popup-menus, which is particularly convenient for touchscreens. The vehicles are fully controllable using the keyboard only, with just a small number and easy to memorise single-key shortcuts, which are also shown in the popup-menus for reference. This makes the console very simple, fast and foolproof to operate even in adverse conditions, like in small boats on open sea.

**Linux CAN**

The development of software for autonomous vehicles often include implementing new device drivers, at the kernel level, for specialized hardware present in the vehicles. An example is the Controller Area Network (CAN), a communications bus commonly used in autonomous vehicles.
to interconnect the various sensors and actuators, but which currently lacks streamlined driver support in modern operating system kernels.

This part of the work presents, first, an Application Programming Interface (API) based on sockets for use in CAN applications such as autonomous vehicles, and second, a complete implementation of a CAN subsystem for the Linux kernel, including the sockets interface, CAN protocols and services, and several CAN device drivers.

There is currently no standard API for CAN software. Device manufacturers sometimes provide drivers for the devices they sell, and each driver usually has a different API. This makes it impossible to develop CAN applications that are device-independent. There are projects developing drivers that support multiple CAN devices using a common API, but often the API is operating system specific, which means that CAN applications developed with that API would be non-portable. Other projects implement drivers that use a standard API, typically the open, read, write, ioctl functions, but in this case higher-layer CAN protocols have to be implemented in user-space, which incurs a severe performance penalty caused by the constant user-space-to-kernel-space-and-back memory copies and context switches. It is therefore proposed an API for CAN that uses the Berkley socket paradigm, for being easy-to-use, flexible, efficient and portable. The socket, bind, connect, send, recv system calls are used to transfer CAN frames to and from the kernel in an operating system independent way. It if defined a PF_CAN for the CAN socket protocol family identifier, and a sockaddr_can data structure to pass the arbitration field of CAN frames, both standard format frames and extended format frames. Higher-layer CAN protocols developed at the DSOR laboratory were also specified, by defining a CANPROTO_DSOR CAN socket protocol identifier, and the various sockaddr data structures for each variant of the DSOR protocols.

A complete implementation of a CAN subsystem for the Linux 2.6 kernel, more precisely 2.6.30, the latest stable release at the time of writing, is provided. The constituting modules are depicted in Figure 16. The pf_can module is the CAN socket protocol family multiplexer. It accepts the socket system calls from user space with the PF_CAN identifier and handles them to the appropriate CAN protocol module. The can module implements the CAN 2.0 protocol in both specification parts A and B, standard and extended format frames, respectively. The dsor\* modules implement the DSOR higher-layer CAN protocols and services, that provide for sending larger than 8 byte CAN messages, relaying UDP/IP packets through CAN, and a standalone and a redundant time services for other CAN nodes. The sja1000 and i82527 are drivers for the two most popular CAN controllers on the market, the Philips SJA1000 and the Intel 82527. Because these are present in devices with different hardware interfaces, these drivers were implemented using a hardware abstraction layer that simply requires subclassing drivers to implement the low-level hardware access functions. The sja1000-isa and pcan-pci modules are drivers that use the sja1000 module, and merely implement the hardware access to the SJA1000 using the ISA and PCI interfaces, respectively. The tscan1 is also a driver that uses the sja1000 module to interface with a PC/104 card that uses a Xilinx PLD to interface the SJA1000 controller. The i82527-isa module similarly implements the ISA interface between the i82527 module and the 82527 controller on an ISA bus. Finally, the canloop driver implements a virtual CAN device that allows simulating a CAN bus in software, for testing purposes, without requiring any CAN hardware.

The performance of the Linux CAN subsystem was evaluated by measuring the time spent by the kernel in all interrupt, software interrupt and process contexts under worst case conditions, which happen when receiving extended format frames with maximum data length, back to back, at full 1Mbps bus rate. To minimize the interference of concurrent kernel activity on the results, a dedicated evaluation system was put together, with the same specifications of the DelfimX PC/104 computer, running the Linux kernel with the CAN subsystem and a single user process to read the received frames. At 1Mbps maximum bus rate, extended format frames with the maximum data length (8 bytes) are transmitted in 130\(\mu\)s, including the 2-bit minimum interframe spacing. The Linux CAN subsystem takes a total of less than 48\(\mu\)s to completely handle and deliver a CAN frame to user space.
Conclusions

The full process of developing the software for an autonomous vehicle was addressed. Step by step, each problem was introduced and simple solutions were given. Simplicity is the key word throughout this work, because a complex software architecture is a wrong software architecture.

The software architecture for autonomous vehicles herein developed included, from high to low, the implementation of a software-based console for monitoring and controlling multiple autonomous vehicles simultaneously, proper selection of network protocols for communicating with the vehicles, the implementation of the vehicles’ critical software, managing multiple sensors with different rates, data logging, profiling, software simulation of autonomous vehicles, integrating all the necessary software for creating the file system of an autonomous vehicle, choosing, configuring and building an operating system kernel optimized for autonomous vehicles, the boot process, and a complete implementation of Controller Area Network for the Linux kernel, both commonly used in autonomous vehicles, that includes the kernel socket interfaces, the CAN protocols and CAN device drivers.

The software architecture was then implemented and put to the test in the DelfimX autonomous catamaran. Real profiling data was gathered from the DelfimX’s sea tests, which demonstrated extremely good performance results. The DelfimX has since endured several other missions at sea, and this software architecture has proven very efficient and reliable during operation. Most importantly, it has proven to be very easy to implement, maintain, test, upgrade, during the whole development process. Given this success, the Delfim autonomous catamaran, the predecessor of DelfimX, also had its software re-implemented using this software architecture.

From the software programmer’s perspective, this work covered all specialties: high-level object-oriented programming of graphical user interface applications, programming size-optimized network servers, low-level programming of time-critical user applications, profiling, software integration, kernel building and kernel-level programming of device drivers and network protocols.