How to Build a Timely Computing Base using Real-Time Linux

António Casimiro
casim@di.fc.ul.pt
FC/UL[*]

Pedro Martins
pmartins@di.fc.ul.pt
FC/UL

Paulo Veríssimo
pjv@di.fc.ul.pt
FC/UL

Abstract

In a recent paper we introduced a new model to deal with the problem of handling application timeliness requirements in environments with loose real-time guarantees. This model, called the Timely Computing Base (TCB), is one of partial synchrony. From an engineering point of view, it requires systems to be constructed with a small control part, a TCB module, to protect vital resources with respect to timeliness and to provide basic time related services to applications. Although many different instantiations of systems with a TCB can be envisaged, we have chosen to implement a TCB using PC hardware running the Real-Time Linux operating system over a Fast-Ethernet network. This paper describes the experience gained during the implementation process and shows that it is possible to construct a TCB without the need for special software or hardware components. The problem of achieving real-time communication under RT-Linux is also discussed: we describe the port we have done of a Linux network driver to RT-Linux, explaining the required modifications to allow predictability.

1 Introduction

In the recent years we have assisted to a very quick improvement of PC hardware that made possible the appearance of increasingly demanding PC-based applications. While most application requirements, such as processing power, memory or multimedia capabilities can be fulfilled just by increasing the available computing resources, other, like timely or real-time behavior, still require other solutions. This is the case of embedded applications for shop-floor control, with strict requirements for timeliness and predictable behavior. Although PC hardware could be used in this case, it would require real-time operating systems that are typically expensive, proprietary and, in some cases, unable to interact in open environments. The problem of using shared and low cost infrastructures in environments where applications have heterogeneous timeliness requirements (synchronous applications cohexisting with asynchronous ones), has to be dealt in the context of system and programming models, assuming unreliable or unpredictable environments, identifying what is critical to timeliness and providing solutions when possible.

In a recent paper we introduced a new model to deal with the problem of handling application timeliness requirements in environments with loose real-time guarantees. This model, called the **Timely Computing Base (TCB)**, assumes that systems, however asynchronous they may be, and whatever their scale, can rely on services provided by a special module, the TCB, which is timely, that is, synchronous. Furthermore, these services can be provided in a distributed way.

In terms of infrastructures, the recent public availability of Real-Time Linux, an operating system with real-time capabilities, designed to operate on PC hardware, makes it a potentially good and cheap platform to implement a TCB. On the other hand, the characteristics of switched Fast-Ethernet networks, including its wide diffusion, low price and potential for, under certain circumstances, providing real-time guarantees, makes it a potentially good platform to implement a TCB.

Given the above, we started the implementation of a TCB on RT-Linux, connected by a 100Mbit Ethernet network. This paper describes the experience gained during the implementation process and shows that it is possible to construct a TCB without the need for special software or hardware components. We discuss some issues that may affect an effective real-time behavior of RT-Linux. We also discuss the problems of real-time communication under RT-Linux and describe the port we have done of a Linux network driver to RT-Linux.

The paper is organized as follows. Section 2 motivates this work and describes some related work. Then, in section 3 the TCB model and services are described. Section 4 presents the essential characteristics of RT-Linux, and discusses the fundamental problems to implement TCB services. The real-time communication aspects and the network driver for RT-Linux appear in section 5. Section 6 is devoted to the presentation of some experimental results. We conclude the paper in section 7.

2 Motivation and Related Work

The problem of handling application real-time requirements when the environment is unpredictable or unreliable is known to be a complex task. In fact, it requires a different reasoning about the systems, not in terms of hard

[*] Faculdade de Ciências da Universidade de Lisboa. Bloco C5, Campo Grande, 1749-016 Lisboa, Portugal. Navigators Home Page: http://www.navigators.di.fc.ul.pt. This work was partially supported by the FCT, through projects Praxis/P/EEI/12160/1998 (MICRA) and Praxis/P/EEI/14187/1998 (DEAR-COTS), and by the EC, through project IST-1999-11583 (MAFTIA).

or soft real-time but it terms of *correct* real-time. Given that real-time or timeliness requirements are expressed by *timing specifications*, correctness of execution has to rely on the ability to (timely) detect *timing failures*, so that appropriate safety measures can possibly be taken.

Some previous papers have proposed system models to tackle the partial synchrony problem. This includes the timed-asynchronous model, where hardware clocks provide sufficient synchronism to make decisions such as ’detection of timing failures’ or ’fail-safe shutdown’ [6], the quasi-synchronous model, where some parts of the system have enough synchronism to perform ’real-time actions’ with a certain probability [17], and the work on partial synchrony presented in [7, 8]. All these papers have in fact motivated the idea behind the work on the TCB model [19]: the search of a generic paradigm for systems with uncertain temporal behavior.

The significance of the TCB model in the context of factory environments is particularly important when the trend is to use low-cost components, shared by many applications. Given the unpredictability of such an environment, those applications with real-time requirements will be subject to timing failures. Therefore, the implementation of several applications with different synchrony requirements in a shared environment can benefit from using a model that ’knows’ about timing failures and that provides solutions to deal with them. However, the model is only useful if its assumptions hold with sufficient coverage [15] for a given purpose. For instance, since the TCB model assumes that a small part of the system has synchronous properties, any implementation of the TCB should use components that provide those properties.

A system with a TCB is not bound to a particular hardware or network. In fact it can assume many different forms. For instance, a real-time embedded system, with a watchdog circuit that stops the system when some deadline is missed, is a good example of a rudimentary TCB, with no processing capabilities, but with ’more synchronous’ properties than the rest of the system and with the possibility to observe its timeliness and act upon failures. This was the approach taken in the MARS system [13] to implement some of the safety measures. Another example of such a dual system is described in [9].

In this work we use PC hardware due to its low cost, availability and potentialities. When using PC hardware for real-time purposes, an adequate operating system must be chosen. QNX [12], LynxOS, VxWorks and pSOS+ are all examples of high-end RTOSs that offer good development environments, are reliable and, above all, provide all the necessary real-time features. However, they are commercial products, provided at high costs. On the contrary, the real-time extension to the Linux operating system, RT-Linux [3], is free software, easily available and increasingly popular. It is obviously a very attractive choice for the purposes of our work. For the communication infrastructure it would be possible to choose a network with specific characteristics for real-time operation, like ATM [16] or CAN [1], but to our evaluation purposes we use instead a Fast-Ethernet [2] network. While the former networks are more expensive and used for particular applications, the later is widely used and, within certain conditions, may also provide the required predictable behavior.

3 A Generic Model for Timely Computing

Due to lack of space, this section only presents a brief overview of the Timely Computing Base model, its services and interfaces. A complete and detailed description of these issues appears in [18] and [19].

3.1 The Timely Computing Base Model

A system with a TCB is divided into two well-defined parts: a *payload* and a *control* part. The generic or *payload* part prefigures what is normally ’the system’ in homogeneous architectures. It exists over a global network or *payload* channel and is where applications run and communicate. The *control* part is made of local TCB modules, interconnected by some form of medium, the *control* channel. Processes p execute on several sites, making use of the TCB whenever appropriate. Figure 1 illustrates the architecture of a system with a TCB.

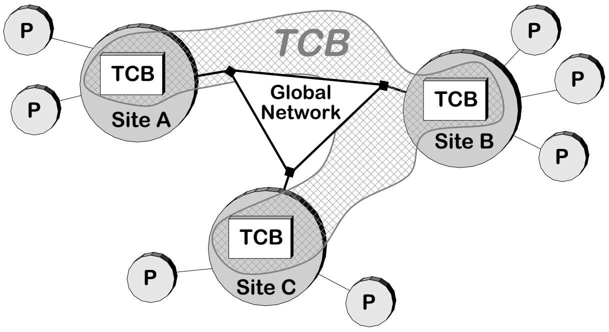

Figure 1. The TCB Architecture.

Concerning the payload part, the important property is that the system *can have any degree of synchronism*, that is, if bounds exist for processing or communication delays, their magnitude may be uncertain or not known. Local clocks may not exist or may not have a bounded rate of drift towards real time. The system is assumed to follow an omissive failure model, that is, components *only do timing failures*— and of course, omission and crash, since they are subsets of timing failures— no value failures occur.

In the control part, there is one local TCB at every site, fulfilling the following construction principles (in section 4 we devote a special attention to these principles):

Interposition - the TCB position is such that no direct access to resources vital to timeliness can be made in default of the TCB

Shielding - the TCB construction is such that it itself is protected from faults affecting timeliness

Validation - the TCB functionality is such that it allows the implementation of verifiable mechanisms w.r.t. timeliness

TCB modules are assumed to be fail-silent, that is, they only fail by crashing. Moreover, it is assumed that the failure of a local TCB module implies the failure of that site.

In terms of synchrony, the TCB subsystem preserves, by construction, upper bounds on processing delays (property **Ps 1**), on the drift rate of local TCB clocks (**Ps 2**) and on the delivery delay of messages exchanged between local TCBs (**Ps 3**).

Given the above set of construction principles and properties, a TCB can be turned into an oracle for applications (even asynchronous) to solve their time related problems. To accomplish this, a set of minimal services has to be defined, as well as the payload-to-TCB interface.

3.2 TCB Services

In order to keep the TCB simple, the services defined are only those essential to satisfy a wide range of applications with timeliness requirements: ability to measure distributed durations with bounded accuracy; complete and accurate detection of timing failures; ability to execute well-defined functions in bounded time. Table 1 presents an informal summary of these services.

Timely Execution

TCB 1 Eager Execution: *Given any function f with an execution time bounded by T, the TCB is able to execute f within T from the execution start instant*

TCB 2 Deferred Execution: *Given any function f and a delay D, for any deferred execution of f triggered at real time t, the TCB will not execute f within D from t*

Duration Measurement

TCB 3 *Given any two events occurring in any two nodes at instants t_s and t_e, the TCB is able to measure the duration between those two events with a known bounded error. The error depends on the measurement method.*

Timing Failure Detection

TCB 4 Timed Strong Completeness: *Any timing failure is detected by the distributed TCB within a known interval from its occurrence*

TCB 5 Timed Strong Accuracy: *Any timely action finishing no later than some know interval before its deadline is never wrongly detected as a timing failure*

Table 1. Basic services of the TCB.

3.3 Providing Adequate Programming Interfaces

Beside defining essential services to be provided by the TCB, it is very important to design a programming interface to allow potentially asynchronous applications to dialogue with a synchronous component. A relevant aspect to understand what can be done, is that applications can only be as timely as allowed by the synchronism of the payload system. The TCB, although being a synchronous component, does not make applications timelier, it only provides the means to detect how timely they are. However, since it can detect timing failures, it may execute timely contingency plans, such as timely fail-safe shutdown. This is very relevant in the context of shop-floor control applications, because it allows the execution of timely and orderly safe procedures when a timing failure occurs. Another important aspect is that nothing obliges an application to correctly use, or use at all, the TCB capabilities. Applications are autonomous entities that take advantage of the TCB by construction. They typically use it as a pacemaker, letting it assess (explicitly or implicitly) the correctness of past steps before proceeding to the next step.

The interface summarized in Table 2 makes a bridge between a synchronous environment and a potentially asynchronous one. Some examples of how to use this interface can be found in [19].

Duration Measurement

`timestamp ← getTimestamp ()`
`tag ← startMeasurement (start_ev)`
`end_ev,duration ← stopMeasurement (tag)`

Timely Execution (eager & deferred)

`end_ev ← exec (start_ev, delay, T_exec, f)`

Timing Failure Detection

`tag ← startLocal (start_ev, spec, handler)`
`end_ev,duration,faulty ← endLocal(tag)`
`tag ← send (send_ev, spec, handler)`
`tag,deliv_ev ← receive ()`
`dur`$_1$`,faulty`$_1$ ··· `dur`$_n$`,faulty`$_n$ `←waitInfo(tag)`

Table 2. Summary of the API.

4 The Real-Time Linux TCB

Given the above description of the TCB, it is clear that any implementation of a system with a TCB is only viable if at least some parts of the system have synchronous properties. In what follows we analyze the Real-Time Linux (RT-Linux) system and its potential to support the implementation of a TCB.

4.1 RT-Linux System Overview

The effort to provide Unix-like operating systems with real-time capabilities is not recent. Although several approaches exist, the RT-Linux developers followed a low-level one, implementing a real-time kernel underneath the operating system and making Linux itself to run as another real-time task [3]. Since it runs with the lowest priority, it can be preempted at any time by higher priority tasks.

The key mechanism to achieve this behavior is based on a virtual interrupt scheme implemented within RT-Linux. This scheme was designed to address the problem of turning the non-preemptable Linux kernel into a preemptable one. Basically, it consists in modifying two things: a) the interrupt table vectors, so that they point to RT-Linux interrupt service routines instead of Linux ones; b) the macros that Linux uses to disable and enable interrupts, `cli` and `sti`, so that some RT-Linux code is executed in its place. This interposition allows RT-Linux to have a complete control over interrupts and, in particular, to prevent Linux from disabling them for a long, unpredictable time.

Another important aspect of RT-Linux is that it only provides basic services: low-level task creation, installation of interrupt service routines, and queues for communication among the real-time tasks and Linux processes. If any real-time I/O interaction has to be done, it is necessary to design specific drivers for RT-Linux. For instance, in the current TCB implementation we had to redesign the Linux network driver to operate under RT-Linux (see section 5).

Finally, we should mention the existence of specific RT-Linux versions for multiprocessor architectures. These are, however, out of our current concerns, and so will not be considered in rest of the paper.

4.2 Implementing TCB services

In order to preserve the synchrony properties **Ps 1** to **Ps 3**, the TCB module has to be constructed in the real-time part of RT-Linux. More specifically, the parts that deal with the API will reside on the non-real-time domain and only the critical parts (periodic activities and timely executions) will have to be implemented as real-time tasks. But there are a number of problems that have to be solved, to obtain a valid TCB implementation. Namely, it is necessary to find affirmative answers to the following questions:

- Is it possible, under RT-Linux, to accurately predict the execution time of TCB activities, which is needed for a schedulability analysis?
- Can the TCB handle multiple service requests, arriving at unpredictable instants, and still behave timely and provide timely services?
- Is it possible to implement a RT-Linux TCB, following the TCB construction principles (see section 3)?

The remainder of this section discusses these questions, and where appropriate describes the solutions employed in our implementation.

Predictability in RT-Linux

In a system with RT-Linux, where activities belonging to two different synchrony domains have to share the same hardware resources, there are potential problems of interference. In fact, in RT-Linux it is possible to observe a phenomena similar to priority inversion, when an higher priority activity interrupts its execution due to some event related to an asynchronous activity. Furthermore, the number and frequency of these events is unknown, and cannot be bounded. The effect is that the execution time of real-time tasks can not be accurately predicted. We identified two situations where this effect could be observed:

- Data transfers between Linux (asynchronous) devices and the host memory can be made through DMA. These DMA bursts are out of RT-Linux control and have priority over the CPU on the access to the system bus. Therefore, if the CPU is processing a real-time task when a Linux device starts a DMA transfer, the execution will stop until the DMA operation finishes. For example, when a packet arrives at a network interface, the card autonomously starts a DMA transfer to copy the packet into the system memory, interrupting all other activities.
- In RT-Linux systems, hardware devices (including Linux ones) are allowed to raise interrupt requests that are handled by the virtual interrupt layer of RT-Linux. Whatever is done in this layer is not important. The fact is that an handler task is executed, possibly delaying the execution of a real-time activity.

Surprisingly or not, RT-Linux is clearly not a perfect real-time operating system, at least not for generic PC hardware. Consequently, our approach in terms of implementation was: a) in first place, find particular solutions to avoid or minimize the causes of unpredictable behavior; b) live with the unavoidable cases, but employ safety mechanisms that are activated upon the (expected rare) occurrences of timing failures.

The overhead caused by sporadic interrupts can be avoided if they are disabled during the execution of real-time tasks. By doing this, two TCB related (real-time) interrupts are also disabled: the clock interrupt that drives the RT-Linux scheduler and the interrupt generated by the network interface card when TCB packets are received. Fortunately, it is possible to ensure that no clock interrupts have to be raised while interrupts are disabled (considering typical task lenght). Since the RT-Linux scheduler uses a one-shot timer to resume real-time tasks precisely when needed, it is sufficient to construct an admission control module that does not allow tasks to overlap. This module must handle a periodic task (TFD service) and several sporadic tasks (Timely Execution service and failure handlers). Provided that all task periods and latencies are known, the module can decide which new tasks can be admitted. The network interrupt, on the other hand, can occur at any instant. In consequence, real-time actions that should be performed by the interrupt handler will be delayed until the interrupts are enabled. The effects of this delay will be discussed afterwards, in this section.

Regarding the problem of DMA bursts, one of the possible solutions could consist in disabling DMA transfers or modifying the priority assignments to the bus access. Unfortunately, this would require some architectural hardware modifications, which is out of the scope of this work. In fact, this would frustrate the basic idea of using standard PC hardware to implement the TCB. The generic solution to this problem is implicitly provided by the *validation* construction principle. By this principle the TCB will know when a synchrony assumption is violated (e.g. as a result of a DMA burst), and safety measures may possibly be applied. The validation mechanisms are discussed in the answer to the third question.

An approach to alleviate the problem of uncertain execution delays, is to add an extra amount to the maximum task execution time. Note that this amount has always to be added, at least to account for the scheduling delay variability (typical values in RT-Linux, using a 120MHz Pentium based PC, are less than 20μs [3]). In section 6 we provide our own measurements for the scheduling delay, which were used in the implementation.

Handling Application Requests

The TCB does not impose any restriction on the amount or frequency of service invocations. Applications may request TCB services whenever they want. Similarly, they are free to provide whatever service parameters they want. The problem is to conciliate this flexibility with the limited processing capacity of the TCB.

A simple but effective solution, is to implement an admission control layer in the TCB interface to filter the requests that cannot be served or that provide incorrect parameters. The admission procedures, although being part of the TCB interface, have to be executed in the payload part of the system to avoid the possibility of an uncontrolled access to the real-time part. In the case of `startExec`, `startLocal` or `send` requests, which may require real-time tasks to be executed, the admission module has to know the start instant and the deadline of these tasks. If the service can not be provided, the application will be informed of the denial reasons.

But the admission control layer does not solve all the interfacing problems. In the above three services, the real-time function specified by the application through parameter `f` or `handler` (see Table 2) cannot be verified at run-time. In our RT-Linux implementation, every real-time function is previously loaded into the kernel space (by means of the Linux loadable module facilities) and the application only provides a reference to one of those pre-loaded functions. Each loadable module contains a real-time function and its estimated worst case execution time (WCET). Since the TCB does not inspect the function code, neither verifies the correctness of the provided WCET, it is assumed that users will not intentionally provide incorrect information. Note that the ability lo load a kernel module is only available to privileged or trusted users. Nevertheless, it is possible that unintentional mistakes occur, like the provision of functions with run-time errors or the provision of a WCET smaller than the real one. In the former case, since the error may lead to an uncontrolled and unpredictable system behavior, any solution would require specific fault-tolerance techniques. On the other hand, the provision of a smaller WCET may at worse generate a timing failure. In this case, the TCB self-checking procedures (that we discuss below) will detect the timing failure and execute adequate safety procedures.

In the current implementation, we assume that the user is responsible for the calculation of WCET values. However, it would be possible to develop a TCB external module for off-line code generation and WCET calculation. Such an integrated solution would simplify the development of applications and would solve the above-mentioned failure problems by ensuring the correctness of all parameters.

Enforcing Interposition, Shielding and Validation

Any implementation of a TCB must be ruled by the *Interposition*, *Shielding* and *Validation* construction principles. Fortunately, and perhaps naturally, the RT-Linux architecture greatly simplifies the task. In fact, since RT-Linux is designed to allow the execution of real-time tasks within an asynchronous system, it must control essential resources for timeliness (interposition) and must preserve the real-time behavior no matter what happens in the rest of the system (shielding). Since these two principles are implicitly granted, our concerns in terms of implementation are essentially focused on the validation principle.

The TCB is assumed to be fully synchronous, as per properties **Ps 1** to **Ps 3**. Therefore, it should be built in a way that secures those properties with $\langle bound, coverage \rangle$ pairs adequate to the timescales and criticality of the application. But there is always a risk of deadlines being missed, mainly if sporadic or event-triggered computations take place [5]. To amplify the coverage of the **Ps** properties the TCB employs a few measures that are based on the validation principle. The idea is to transform unexpected timing failures into crash failures, enforcing a fail-silent behavior. The measures are carried out using monitoring mechanisms based on fail-awareness techniques, i.e., techniques that allow a component to realize it has suffered a timing failure [10]. As shown in Figure 2, the role of these mechanisms, to which we call *self-checking mechanisms*, is to observe the interactions between the TCB services and the system hardware resources, detect violations of assumptions, and, if that happens, activate a fail-silence switch.

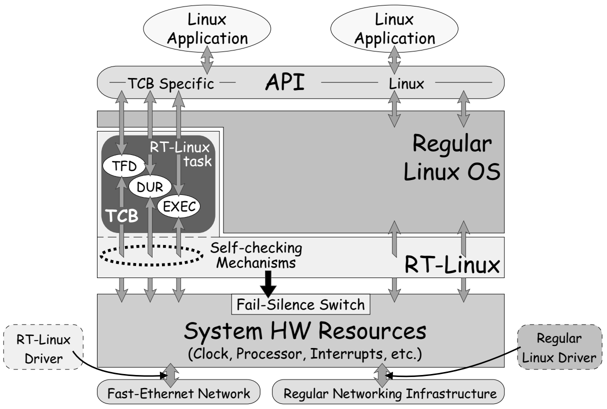

Figure 2. Block Diagram of a RT-Linux system with a TCB

Although it would be possible to implement self-checking mechanisms to observe all three **Ps** properties, we assume that **Ps 2** (clocks) is always valid. Therefore, the implementation only considers self-checking mechanisms for **Ps 1** (processing) and **Ps 3** (communication).

To detect unexpected timing failures on processing delays (typically, executions exceeding their WCET), we use the local clock to measure execution durations. For each time-critical computation starting at instant T_s, with a maximum duration of T_c, the self-checking procedure sets an alarm for the desired deadline ($t_{dead} = T_s + T_c$). If the computation does not end by t_{dead} the alarm trips and causes the immediate activation of the fail-silence switch, crashing the whole site. Note that the time required to execute the self-checking procedures has to be taken into account by the admission control layer.

To detect a violation of **Ps 3**, message delivery delays have to be measured. This is done using a round-trip duration measurement technique similar to the one used to measure distributed durations. The idea is to build a fail-aware broadcast [11] as the basic inter-TCB communication primitive. When a message is not delivered on time, the service raises an exception that causes the immediate activation of the fail-silence switch. Note that since the exception is not raised at the deadline, but when the message is delivered, it is not possible to enforce a timely shut-down in all cases. At worse, the coverage amplification effect will not be observed, being the situation as good as without the self-checking mechanisms.

The measurement of message delivery delays in the fail-aware broadcast requires timestamps to be generated on send and delivery events. To obtain accurate measurements, in our implementation this is done by the network driver, right before transferring a packet to the network interface card, and as soon as a new message is stored in memory (through DMA) and an interrupt is raised. Even though, there is a small latency associated to the execution of the interrupt service routine that depends on the maximum interval during which the interrupts are disabled (while the TCB executes some real-time activity). Fortunately, since task execution times are typically much shorter than message delivery delays, the impact on the measured delivery delay is practically negligible.

5 Implementing Communication Services

In our RT-Linux implementation, the TCB synchronous communication channel is based on a physically different network from the one supporting the *payload* channel. For the *control* channel we have used a switched Fast-Ethernet network, exclusively dedicated to connect local TCBs. The rest of the system is interconnected through a normal 10Mbit Ethernet network. Note that this dual network architecture is not strictly required by the TCB model. It would be possible to use some of the current networks to set up virtual channels with predictable timing characteristics, coexisting with essentially asynchronous channels [4, 16].

Given that we have a dedicated network for the control channel, the problem of achieving predictability is much alleviated. In one hand, since traffic generation is restricted to the TCB, we can easily control the network load. On the other hand, by exploiting some specific characteristics of switched Fast-Ethernet networks, it is possible to eliminate the unpredictability introduced by the Binary Exponential Back-off collision resolution algorithm of Ethernet networks. The solution simply consists is avoiding packet collisions. This can be achieved by connecting each port of the switch to a single station to ensure that at most one adapter may transmit on each direction of the full-duplex link. Therefore, provided that the switch buffering capacity is not exceeded (which would introduce additional, possibly unpredictable delays), it is possible to guarantee a maximum transmission delay for each packet.

However, the end-to-end delivery delay also depends on the time it takes to actually send and receive messages or, in other words, on the actions performed by the network driver. This will be discussed next.

5.1 The TCB Network Device Driver

The standard Linux network driver was not designed for real-time operation. Therefore, to achieve a real-time behavior within the TCB and the RT-Linux kernel itself we had to redesign this driver.

For the port, we used the standard Linux driver for 3COM Ethernet cards (model 3C905b), publicly available as part of the Linux kernel distribution under the name `3c59x.c`. Although most of the original code has remained unchanged, like some functions for booting up, shutting down, or retrieving statistics, the final result is not a generic driver, but one specifically designed to operate under the TCB.

The crucial modifications took place at the interface between the driver and the upper levels. The original Linux driver uses an upcall mechanism to deliver incoming frames to the layer above it. Since this upcall (or message copy) is executed by the driver interrupt service routine, in RT-Linux this would consume real-time resources. Furthermore, since messages arrive sporadically the processing overhead would be unpredictable. The solution is conceptually very simple. When the network interface card receives a new frame it starts an autonomous DMA transfer and signals the driver (with an interrupt) when the transfer completes. Then, the driver interrupt service routine gets a timestamp and acknowledges the interrupt: no copies are done, neither the message contents are analyzed. Therefore, the execution overhead of the interrupt routine can be neglected. The driver simply leaves messages in the buffers, waiting for the upper layer to consume those messages periodically.

This approach requires the consuming rate to be at least equal to the rate of incoming messages. Fortunately, since we have control over the number of messages sent to the network, it is easy to know the rate of incoming messages. The period of the consuming task is calculated in order to prevent buffer overflows and consequent message losses.

Provided that all the above bounds are enforced, this approach allows the construction of a real-time communication service under RT-Linux. In fact, it can be guaranteed that every message is consumed within a bounded amount of time after it has been transmitted. This bound depends on the maximum delivery delay (which, for a given load pattern and message size, can be bounded) and on the period of the consuming task. Note that this period should be made as small as possible, since some TCB parameters (e.g. the maximum failure detection latency) depend on it.

5.2 Device Driver Services

Beside the management services provided by the standard Linux driver, the RT-Linux network driver provides the services for transmitting and receiving messages. The former can be used to reliably broadcast an Ethernet packet to the network and the later is used to consume a packet from the device driver buffer.

The transmission service was designed to operate exclusively in broadcast mode because the TCB always disseminates messages to all other TCBs. When a transmission request is issued, the driver generates a timestamp that is also sent and that will be used to measure an upper bound for the delivery delay.

The reliability of the transmission is obtained by letting the service client specify an assumed network omission degree, and by sending a sufficient number of replicated messages to mask that omission degree. The driver was designed to optimize the replicated transmission: instead of copying a message into several transmission buffers, it maps several descriptors of the transmission table to the same memory buffer. On the receiver side, message replicas are not processed by the driver. This would force message contents to be inspected and, consequently, an increased overhead. Duplicate messages are simply discarded by the TCB. Note that all message replicas share the same send timestamp, and thus the maximum delivery delay is calculated for the last replica.

The reception service is used to read messages from driver buffers. When there are available messages, the service returns all of them along with their reception timestamps. These timestamps, and the send timestamps contained in the messages, will be used by the self-checking mechanism to calculate delivery delays.

6 Experimental Results

This section presents the results of a few experiments that we have conducted with the objective of obtaining an initial intuition of what could be expected, in terms of timeliness, from the RT-Linux system and from the switched Fast-Ethernet network. We are currently performing extensive analysis of the essential timing parameters, and will publish them in a future paper.

The tests were performed using the experimental infrastructure depicted in Figure 3. We used three 500MHz PentiumIII based PCs with the RT-Linux system and a TCB, interconnected by a 3COM SuperStack II Baseline switch. For the measurements we used another PC running a special measurement tool (Event Timestamping Tool) that we have developed. This tool consists of a small kernel booted up from a floppy disk, which executes a simple program to read events from an input (the parallel port) and store corresponding timestamps. The granularity of the timestamps depends on the processor speed, but is typically in the order of a few nanoseconds. A detailed description of this tool can be found in [14].

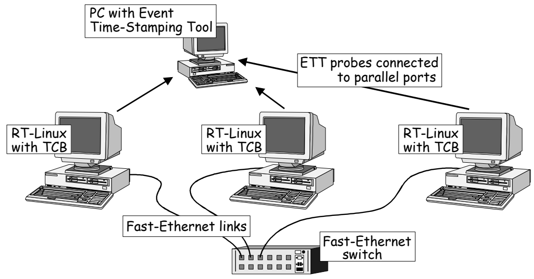

Figure 3. Experimental infrastructure.

However, our first experience was performed without using this tool. The test, to measure the variability of the scheduling delay in RT-Linux, consisted in scheduling a real-time task at some instant t and obtaining a timestamp t_s (from the local clock) as soon as the task was released. We measured a maximum scheduling delay ($t_s - t$) of about 18μs, which is similar to the assumed value of 20μs. The test was performed under heavy system load conditions (generated by payload applications), and for task periods ranging from 100μs to 100ms.

We also measured the scheduling delay variability using the Event Timestamping Tool. We executed a periodic real-time task that did nothing but set up an event on the parallel port. The variability of scheduling delays, for task periods ranging from 100μs to 10ms, was lower than 17μs, without load, and lower than 26μs, under heavy load conditions. This confirms, once again, our expectations.

To analyze the network behavior we performed a few more tests using the measurement tool. The goal was to validate our expectations about the deterministic characteristics of Fast-Ethernet under controlled load conditions and with switch ports allocated to unique nodes. Therefore, the tests consisted in measuring message delivery delays using different message sizes, while increasing the transmission rates until message loss started to be observed (when reaching the *zero-loss* transmission period).

Delivery delays were measured by setting up an event, at the driver level, upon each message transmission or reception. Since all messages are broadcast, each transmission generates three events (a send and two receive events). We performed eight experiences, with all three TCBs transmitting simultaneously, for messages with 100, 300, 680 and 1024 bytes, and transmission periods of 10ms and 1ms. Three more experiences were also conducted to evaluate the *zero-loss* performance of the network. These were for messages with 300, 680 and 1024 bytes. With messages of 100 bytes we were not able to reach the *zero-loss* point, since with only three sending stations too much system resources were required.

The value of 1024 bytes ensures that no receive overruns can possibly happen at the receiver, and that no messages will be lost by this reason. This value was calculated for receiver input FIFOs with 2048 bytes and for a burst of two messages. In each experiment nearly 30000 message delivery delays were measured. Minimum and maximum delays are presented in Table 3.

	Transmission period				
	Zero-loss period				
Frame size	52µs	113µs	168µs	1ms	10ms
100 bytes				42-47	42-49
300 bytes	78-221			84-122	83-90
680 bytes		153-258		153-178	162-177
1024 bytes			221-348	221-359	231-279

Table 3. Message delivery delays (μs).

Although these were not extensive tests, the results are nevertheless sufficient to sort out a few conclusions. Without any further details it is possible to observe that delivery delays can be kept within reasonable intervals, with maximum values not exceeding a few hundred microseconds. It is also possible to observe that the length of the intervals (as well as the upper values) increase with the amount of traffic sent to the network. This is due to higher switching delays, caused by extra buffering time when the switch is operating under heavy load conditions. Note that in the experiences corresponding to the *zero-loss* period, the (input) throughput is near the theoretical maximum network capacity of 100Mbit/s (for example, 2 flows of 300 byte frames with a period of 52 μs correspond to near 93Mbit/s).

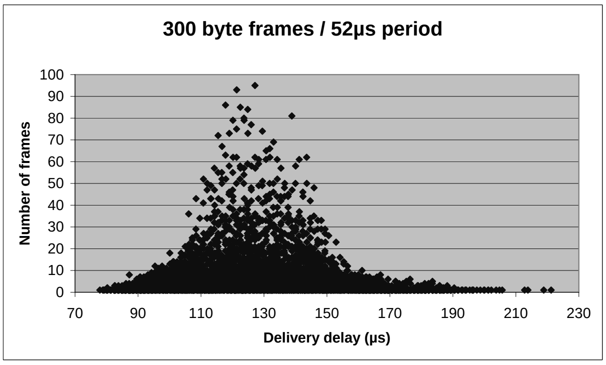

Figure 4. Delivery delay distribution.

In Figure 4 it is possible to observe the distribution of delivery delays for one of the *zero-loss* period experiences. In terms of probability distribution it is clear that there is a peak around the value of 130μs, and that the probability of delivery delay values above 230μs is extremely low. Given the fact that no message loss is observed in these conditions, we can conclude that our initial assumptions about the predictability of Fast-Ethernet networks can indeed be sustainable. Nevertheless, we are currently performing further tests have to consolidate the present results.

7 Concluding Remarks

In this paper we discussed the problem of implementing a TCB without using special hardware or software components. We have used the RT-Linux operating system, PC hardware, and a Fast-Ethernet switched network.

The problem of achieving real-time communication using these components was discussed in detail. The solution combines some features of switched Fast-Ethernet networks with a requirement for bounded network load. We describe our implementation of a real-time network driver for RT-Linux.

Finally, we have presented some experimental results, which were obtained to confirm some basic assumptions. Those results have shown that the RT-Linux system can be used to schedule real-time tasks with a bounded scheduling delay. We were also able to confirm that with a switched Fast-Ethernet network it is possible to avoid network collisions and that upper bounds on message delivery delays can be small and deterministic.

References

[1] Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication, iso 11898, Nov. 1993.

[2] IEEE 802.3u Standard. Local and Metropolitan Area Networks-Supplement - Media Access Control (MAC) Parameters, Physical Layer, Medium Attachment Units and Repeater for 100Mb/s Operation, Type 100BASE-T, Mar. 1995. Supplement to IEEE Std 802-3.

[3] M. Barabanov and V. Yodaiken. Real-time linux. *Linux Journal*, Feb. 1997.

[4] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, Sept. 1997.

[5] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. International Computer Science Series. Addison-Wesley publishers Ltd., 1996.

[6] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, Jun 1999.

[7] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *24th Annual Symposium on Foundations of Computer Science*, Nov. 1983.

[8] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[9] D. Essamé, J. Arlat, and D. Powell. PADRE: A Protocol for Asymmetric Duplex REdundancy. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 213–232, San Jose, California, USA, Jan. 1999.

[10] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 314–321a, Philadefphia, USA, May 1996. ACM.

[11] C. Fetzer and F. Cristian. A fail-aware datagram service. In *Proceedings of the 2nd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, Apr. 1997.

[12] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, USA, Apr. 1992.

[13] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, pages 25–41, Feb. 1989.

[14] P. Martins. Event Timestamping Tool: a simple PC based kernel to timestamp distributed events. DI/FCUL TR 00–4, Department of Computer Science, University of Lisboa, July 2000.

[15] D. Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 386–395, Boston, USA, July 1992.

[16] M. d. Prycker. *Asynchronous Transfer Mode: Solution For Broadband ISDN*. Prentice-Hall, third edition, 1995.

[17] P. Veríssimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39, Winter 1995.

[18] P. Veríssimo and A. Casimiro. The Timely Computing Base. DI/FCUL TR 99–2, Department of Computer Science, University of Lisboa, Apr. 1999.

[19] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN'00)*, New York, USA, June 2000.