FIFO WiDOM: Timely control over wireless links

Viktor Leijon, Per Lindgren and Johan Eriksson

Luleå University of Technology
Department of Computer Science and Electrical Engineering
S-97187 Luleå, Sweden
{leijon,pln,johan.eriksson}@csee.ltu.se

Abstract

We present the idea of scheduling a control network in a first-come first-serve manner, and demonstrate a way to implement this over a wireless link. This achieves real-time guarantees and tightly bounded jitter for control applications. Further, how to design systems using this network is discussed, and the benefits compared to a standard priority MAC protocol are examined through simulations and a prototype implementation. The prototype is implemented using the reactive Timber programming model on a lightweight 8-bit platform.

I. Introduction

Control over wireless links is a rapidly developing field, and presents many new problems in control theory. From a computer engineering point of view, getting a stable communication channel with reliable timing properties is the main complication.

One of the challenges is that wireless nodes are often considered to be more loosely coupled than wired systems; adding and removing nodes may be done more dynamically. If predictable real-time properties are desired in a system such as this, temporal composability [1], as well as considerable flexibility, becomes important.

In previous work we have presented a protocol for FIFO networking [2], and outlined how it could be implemented on-top of the CAN protocol. In this paper we extend it to control a wireless link by combining it with work from Pereira *et al.* [3] on the wireless dominance protocol (WiDOM), creating the FIFO WiDOM MAC protocol.

Our approach gains flexibility and implementation agility compared to what is seen in TDMA systems, and is easy to implement on existing embedded systems with wireless capabilities, with only a modest cost.

Our contributions in this paper are:

- We review the basics of a FIFO based MAC protocol and describe the adaptations needed when implementing it on top of the Wireless dominance (WiDOM) MAC (Section II).
- Simulations of the system, comparing it to a priority based protocol are presented, in Section III.
- In Section IV we give a short introduction to reactive objects and the Timber programming model.
- Finally, we present a prototype implementation of FIFO WiDom for the AVR platform using IR communication. We utilize the programming model of Timber to capture the specification, and a complete protocol implementation is presented. (Section V).

II. Wireless dominance FIFO

A. An overview of FIFO networking

There are many approaches to deciding who gets to use a network at any given time. In a time-triggered (TDMA) network it is decided a priori (or at the set up of a communication session), and at runtime the static schedule decides when a node is allowed to transmit. In an event-triggered system some other method is needed; in the standard Aloha-like protocol chance decides who gets to transmit, and in a priority based system a priority is used to arbitrate between concurrent senders. Many systems also function as hybrids of two or more of the methods.

Our approach is to see the network as a FIFO (Figure 1), and decide who gets to transmit on the network next on the basis of how long a message has been waiting. The message with the longest accumulated waiting time is always transmitted next. If there is a fixed speed at which items are removed from the queue (δ time units per item), and a maximum length (N) of the queue, there will also be a maximum time to send a message.

Fig. 1. A FIFO for messages

More details on the principles can be found in our original article [2], which also includes reasoning about how to design systems with the paradigm, calculating the queue length N , and how to obtain composability. That work carries over with little or no modification to FIFO WiDOM.

B. Using WiDom for FIFO networking

Dominance protocols in general have become very popular in wired control networks, in particular in the CAN network [4], which is particularly common for automotive applications. More recently, Pereira *et al.* [3] have shown how to extend this to wireless networks in the form of the WiDOM protocol.

Analogous to our adaption of the FIFO principle to the CAN bus [2], we can extend WiDOM so that the network behaves like a FIFO as described in section II-A. We explain the principles and parameters of the FIFO WiDom adaptation.

In summary, dominance protocols are based on each message type having a static priority, and any collisions being resolved non-destructively, by an arbitration phase using the priorities of the colliding messages.

In our examples we will assume that 6 priority bits are used in the WiDOM implementation, but this can be extended if more nodes are required. In our prototype implementation (Section V) we have $\delta = 29.12$ ms.

We choose to divide the priority into two parts (Figure 2):

a) *Waiting time*: is the number of arbitration rounds that the node has lost. This value is reset whenever the node is allowed to transmit a message by winning an arbitration.

b) *Node id*: must be unique within each system and is used to break ties when two or more nodes have the same waiting time.

Fig. 2. Overview of a protocol frame.

C. Designing with FIFO WiDOM

The FIFO WiDOM behaves like a FIFO in the sense we explored in Section II-A, therefore the methodology outlined in our previous work [2] can be used to design a system which meets a given set of time constraints.

The distribution of bits between node id and waiting time depends on the distribution between queue slots and nodes. For instance, we may assign 3 bits for node identifier and 3 for waiting time, and thus get $2^3 = 8$ as the maximum number of slots in the queue. When using the maximum queue length the delay is $d = N\delta = 8\delta$ (which is 233 ms in our prototype).

D. Controlled jitter

Jitter control is considered to be especially important for control loops [5], where use of a time invariant control law is generally preferred. We propose a solution for situations where low jitter is preferable over early arrival of messages.

There is a maximum send time, δ , and we design our system with a pre-defined maximum queue size N . In our implementation the actual waiting time (the number of arbitration cycles the message has lost) is a part of the message. This means that if the waiting time portion of the message priority is p , we can always wait $(N - p)\delta$ to give all messages similar delivery times. The worst case jitter is bound by δ plus the maximum deviation in packet send times.

Fig. 3. Send time distributions for two different traffic intensities

III. Simulations

To compare the FIFO approach to a more traditional priority based communication scheme we set up a series of simulations.

In the simulations we created a system with two communicating nodes to simulate a simple event loop, and a number of nodes that produce background traffic. The communicating nodes wait for a message to arrive, and when they have received a message they wait one packet time (to simulate performing some kind of computation) before sending it back. The background nodes each send messages, using an exponential distribution with varying mean intervals between sending. The system was simulated for 100000 packet times.

The number of background nodes was varied, and the case for ten generators of background traffic is studied in particular.

A. Simulation of the FIFO Mac

Setting the mean send time for the background traffic to 10, the average time to send was 3.06 units with a standard deviation of 1.50. Decreasing the interval for the background servers to 5, the average delay increases to 6.43 and the standard deviation is 1.47.

It is worth noting that the maximum delay is 10 in all cases, and that the jitter minimization technique outlined in Section II-D could be used to decrease the jitter even further. For instance, by delaying all packets until 6 packet times have passed in the first example decreases the standard deviation to 0.17.

A plot of the waiting time distribution is shown in Figure 3. We can see that the delays are highly dependent

Priority	Average delay	Max delay	std. dev
0	5.19	72	6.55
5	3.22	39	3.29
10	2.05	16	1.51
15	1.35	8	0.68
20	1.00	2	0.02

Fig. 4. Priority MAC, inter-arrival time 10

Priority	Average delay	Max delay	std. dev
0	225.06	1434	225.88
5	25.77	231	30.40
10	4.98	46	4.37
15	1.80	14	1.16
20	1.00	1	0.0

Fig. 5. Priority MAC, inter-arrival time 5

on the traffic load on the network, with increasing load the response time grows. The delay never goes over the maximum queue length.

B. Simulation of a priority based MAC

In this simulation the 10 background processes are assigned even priorities from 2 to 20. The processes are given priorities from 0 to 21. The results with inter-arrival time set to 5 is in Figure 5 and for 10 in Figure 4

Figure 6 shows priority versus average delay (with standard deviation as error bars) for the different traffic intensities.

Fig. 6. Average delay as a function of the priority, for two different traffic intensities

Fig. 7. A comparison of FIFO and priorities for varying traffic intensities.

C. Comparison

We compared the FIFO MAC and the priority mac at various traffic intensities, letting the interval for the background tasks vary from 1 to 20 packet times. The result is in Figure 7. While the FIFO and priority buses behave similarly for reasonable situations, the priority based traffic has unbounded delays and bad average case behavior when there is a lot of higher priority traffic.

We conclude that care is required when choosing priorities in a priority based scheme, and that selecting the wrong priorities can lead to undesirable consequences, in particular for systems with high worst case load. When constructing a system with priorities, it is important to perform a schedulability analysis [6].

The FIFO based approach has a worst case waiting time that is linear to the number of nodes, preventing starvation in the system, but gives the same guarantees to all messages in the system. In the original paper [2] we present extensions to combine priority levels with the FIFO semantics.

IV. Timber

Embedded real-time systems are naturally described as time-bound reactions to external events, a view supported natively in the high-level programming and systems modeling language Timber [7]. The engineering perspectives of the Timber design paradigm are further elaborated in [8].

In this paper, we implement the dominance protocol using the programming model of Timber, through a C interface (*tinyTimber*, *tT*) to a minimalistic, robust, and

portable Timber run-time kernel. This allows for comprehensive and efficient implementation with predictable memory and timing behavior. The application interface of *tT* features the following subset of Timber:

- Concurrent reactive objects
- State protection
- Synchronous and asynchronous messages
- Deadline scheduling

The Timber run-time model utilizes deadline scheduling directly on basis of programmer declared event information. In short:

- *Objects and concurrency*: The concurrent and object oriented models go hand in hand. An object instance executes in parallel with the rest of the system, while the state encapsulated in the object is protected by forcing the methods of the object instance to execute under mutual exclusion. This implicit coding of parallelism and state protection coincides with the intuition of a reactive object. Furthermore, all methods in a Timber program are non-blocking, hence a Timber system will never lack responsiveness due to dependencies on events that are yet to occur.
- *Events, methods and time*: The semantics of Timber conceptually unifies events and methods in such a way that the specified timing constraints on the reaction to an event can be directly reused as run-time parameters for scheduling the corresponding method. The permissible window for method execution is defined by the *baseline* (the absolute point in time for event release), together with the relative *deadline*. Chain reactions are expressed through relating future events to the locally fixed baseline, and will be free from jitter induced by the actual scheduling.

On the top level, a *tinyTimber* application consists of the implicit root object and its methods, which are the functions installed for handling interrupts and the reset signal (main in C). The state of the root object is the state of the globally declared C variables. To impose more structure, the root state can be further partitioned into objects of their own, whose methods are run under supervision by the scheduler. Method calls crossing object boundaries must never bypass the kernel primitives.

A *tinyTimber* application is compiled and linked with the *tT* kernel for a target architecture using a C compiler suite such as gcc. For a bare-chip target, the executable image will not rely on additional libraries, although libraries may be incorporated as long as they do not violate the run-to-end assumption of *tT*.

V. Implementation

Fig. 8. Measurement of pulse detection delay

Fig. 9. Schematic of the receiver module

A. Hardware

To implement the protocol a board with an ATmega169 from ATMEL is used. The transmitter on the unit is an infrared diode that emits light at a wavelength of 940nm. The diode is modulated with a 38kHz square wave, generated by the hardware using a 8bit timer. A logic 1 is represented by a carrier and 0 as no carrier. The microcontroller platform is powered from a CR2450 lithium cell.

Receiving the modulated signal is done by using a infrared receiver module that detects the presence of the 38kHz carrier. The module signals a detected carrier by pulling the output signal low.

A measurement of a the 38kHz signal, modulated by 1ms pulses is shown in Figure 8. The time delay from the presence of a carrier to a low level on the receiver module is measured to approximately 250μs.

A schematic showing the connection of the diode and receiver module is shown in Figure 9.

The ATmega169 used has a power consumption of 4 mA in active mode and 1.4 mA in idle. The microcontroller is in idle mode most of the time during both sending and receiving, this brings the average current consumption to 1.5mA. The maximum power consumption of the infrared receiver module is 2 mA. On a single cell,

this gives an estimated lifetime for a system in receive mode of approximately one week, or a system sending over 8 million 16-bit frames.

B. Timber objects

The depicted code gives a complete implementation, besides of macros for tT kernel interface (TT_*) and IR hardware (IR_*). The protocol is implemented by an `irDom` object, (Listings 1, 3), which transmits and receives application data (Listing 2) over the IR interface.

The `irDom` object interface defines object structure, initiation mechanism, and externally accessible methods, (Listing 1). In `appObj` object implements a listener to the `irDomObj` echoing incoming data after one FRAME, (Listing 2). `PINCHANGE_INTR(void)` implements the handler for the external interrupt caused by the IR receiver on carrier change, (Nodes B and C, Time 0, Figure 2). `GETTIME()` may utilize available hardware “timestamping” (input capture) of interrupts, to improve timing accuracy. Accordingly, the message `TT_ASYNC(timestamp, SLACK, ...)` will be immediately due for execution, with the eligible execution window of `timestamp...timestamp+SLACK`. `irDomRecSync` will disable carrier change interrupts, and if data pending, skipping the *start* bit and entering the tournament at Time 1 (Node B), or skipping the header bits and start receive at Time 7.5 (Node C).

In `irDomTournament` we first check if we have survived through the complete header (won the tournament), if so enter `irDomTransmit` (Node B, Time 7). We proceed by checking if our header bit is dominant, if so we activate our carrier and proceed with the next bit in order (Nodes A and B, Time 1). On a recessive bit, we deactivate our carrier and issue a listen for carrier halfway into the period (Nodes A and B, Time 2). In `irDomRecessive` we check for incoming carrier, if so we have lost the tournament (Node A, Time 3.5) and skip remaining header bits before start receiving data (Node A, Time 7.5). If we have data pending, we increase our FIFO priority (Node A, Time 3.5). In `irDomReceive` (Nodes A and C, Time 7.5...14.5) and `irDomTransmit` (Node B, Time 7...14) we receive and transmit data correspondingly.

When received finished, data is sent to the installed “callback” (Nodes A and C, Time 14.5). When transmit finished, we deactivate our carrier (Node B, Time 15). At Time 15.5, we clear pending carrier change interrupts and enable new. If data pending we schedule a new packet send (Node A, Time 15.5).

It is worth to note that the `irDomObj` object will be responsive to `irDomSend` events throughout the complete protocol execution. However, the current implementation does not support buffering of more than one data element,

```

#define PERIOD          MSEC(2 * 0.889)
#define SLACK           (0.2 * PERIOD)

#define NRPRIOSTAT      3 // static priority, ID
#define NRPRIOFIFO      3 // 0 for static priorities only
#define NRPRIO          (NRPRIOSTAT + NRPRIOFIFO)
#define NRDATA          8
#define SLOT            ((NRDATA + NRPRIO + 2) * PERIOD)

typedef struct {
    TT_OBJ o; // Object state
    TT_OBJ* listenObj; // Listener Object
    TT_METH listenMeth; // Listener Method
    BOOL busy;
    BOOL dataPending;
    short prio; // [NRPRIOBITS]
    short data; // [NRDATABITS]
    short rec; // [NRDATABITS]
} IrDom;

#define initIrDom(obj, meth) {TT_INIT_OBJ(), obj, meth, FALSE, FALSE}

void irDomPrio(IrDom* self, int arg);
BOOL irDomSend(IrDom* self, int arg);
void irDomRecSync(IrDom* self, int arg);

```

Listing 1. irDom.h

```

typedef struct {
    TT_OBJ o;
    idDom* ir;
    ...
} App;

void appListen(appObj* self, int arg) {
    TT_ASYNC(PERIOD, SLACK, self->ir, irDomSend, arg + 1);
    ...
}

extern IrDom irDomObj;
App appObj = {TT_INIT_OBJ(), &irDomObj, ...};
IrDom irDomObj = initIrDom(&appObj, appListen, IR_MAC);

PINCHANGE_INTR(void) {
    TT_SAVECONTEXT();
    Time timestamp = TT_GETTIME();
    ...
    IR_DISABLE_PINCHANGE_INTR();
    TT_ASYNC(timestamp, SLACK, &irDomObj, irDomRecSync, IR_IS_CARRIER());
    TT_SCHEDULE();
    TT_RESTORECONTEXT();
}

int main(void) {
    TT_INIT(); // kernel init
    IR_INIT(); // IR hardware initialization
    IR_CARRIER_OFF();
    IR_ENABLE_PINCHANGE_INTR();
    TT_IDLE(); // enter kernel idle
}

```

Listing 2. root.c

and will return FALSE if data is already pending. Furthermore, due to space limitations, the example code does not implement header information for received packages. A complete implementation is available from the authors.

VI. Related work

Mosley performed some early work on general FIFO networking [9] in the context of the tree-algorithms for retransmitting proposed by Capetanakis [10].

The work on WiDOM is due to Pereira, Andersson and Tovar [3], and includes the extension of response time analysis to that protocol.

There have been several approaches to real time guarantees for CAN, in particular the work of Tindell *et al.* [6] provided a foundation based on scheduling theory for

```

void irDomPrio(IrDom* self, int arg) {
    self->prio = arg;
}

BOOL irDomSend(IrDom* self, int arg) {
    if (self->dataPending) {
        return FALSE;
    }
    self->data = arg;
    self->dataPending = TRUE;
    self->prio &= (1 << NRPRIOSTAT) - 1; // (enter FIFO)
    irDomSendSync(self, 0);
    return TRUE;
}

void irDomSendSync(IrDom* self, int arg) {
    if (self->busy) {
        self->busy = TRUE;
        IR_DISABLE_PINCHANGE_INTR();
        IR_CARRIER_ON();
        TT_ASYNC(PERIOD, SLACK, self, irDomTournament, NRPRIO);
    }
}

void irDomRecSync(IrDom* self, int arg) {
    if (self->busy) {
        return;
    }
    self->busy = TRUE;

    if (self->dataPending) {
        TT_ASYNC(PERIOD, SLACK, self, irDomTournament, NRPRIO);
    } else {
        TT_ASYNC((NRPRIO + 1.5) * PERIOD, SLACK, self, irDomReceive, NRDATA);
    }
}

void irDomTournament(IrDom* self, int arg) {
    if (arg == 0) {
        irDomTransmit(self, NRDATA);
        return;
    }
    arg--;
    if (self->prio & (1 << arg)) { // dominant bit
        IR_CARRIER_ON();
        TT_ASYNC(PERIOD, SLACK, self, irDomTournament, arg);
    } else { // recessive bit
        IR_CARRIER_OFF();
        TT_ASYNC(0.5 * PERIOD, SLACK, self, irDomRecessive, arg);
    }
}

void irDomRecessive(IrDom* self, int arg) {
    if (IR_IS_CARRIER()) { // carrier, tournament lost
        TT_ASYNC((arg + 1) * PERIOD, SLACK, self, irDomReceive, NRDATA);
        if (NRPRIOFIFO) { // FIFO mode
            short fifo = ((self->prio >> NRPRIOSTAT) + 1) & ((1 << NRPRIOFIFO) - 1);
            if (fifo == 0) { // fifo overrun
                TT_PANIC("irDomRecessive, _FIFO_overrun", 0);
            }
            self->prio = (self->prio & ((1 << NRPRIOSTAT) - 1)) | (fifo << NRPRIOSTAT);
        }
    } else { // no carrier, survived recessive bit
        TT_ASYNC(0.5 * PERIOD, SLACK, self, irDomTournament, arg);
    }
}

void irDomReceive(IrDom* self, int arg) {
    arg--;
    if (IR_IS_CARRIER()) { // carrier, databit = 1
        self->rec |= 1 << arg;
    }
    if (arg == 0) { // finished
        self->busy = FALSE;
        TT_ASYNC(PERIOD, SLACK, self, irDomInterPackage, arg);
        TT_ASYNC(PERIOD, SLACK, self->listenObj, self->listenMeth, self->rec);
    } else {
        TT_ASYNC(PERIOD, SLACK, self, irDomReceive, arg);
    }
}

void irDomTransmit(IrDom* self, int arg) {
    if (arg == 0) { // finished
        IR_CARRIER_OFF();
        self->busy = FALSE;
        self->dataPending = FALSE;
        TT_ASYNC(PERIOD * 0.5, SLACK, self, irDomInterPackage, 0);
        return;
    }
    arg--;
    if (self->data & (1 << arg)) {
        IR_CARRIER_ON();
    } else {
        IR_CARRIER_OFF();
    }
    TT_ASYNC(PERIOD, SLACK * PERIOD, self, irDomTransmit, arg);
}

void irDomInterPackage(IrDom* self, int arg) {
    IR_CLEAR_PINCHANGE_INTR();
    IR_ENABLE_PINCHANGE_INTR();
    if (self->dataPending) {
        irDomSendSync(self, arg);
    }
}

```

Listing 3. irDom.c

calculating worst case response times in a network where each message has a fixed minimum period. There are some recent improvements to this work by Bril, *et al.* [11].

In the time-triggered realm, a lot of work has been done on guaranteeing timing properties. One early time-triggered effort was the time-triggered protocol [12] and the time triggered architecture [13]. Kopetz and Obermaisser has stressed the importance of temporal composability [1]. In trying to understand the differences between CAN and TTA, it is valuable to read his study [14]. There has also been time-triggered variants on CAN, most notably TTCAN [4] and FTT-CAN [15].

The technology which most obviously resembles the FIFO approach is that of the timed token network [16], where a token is sent around the network. If "being first in the queue" is seen as an imaginary token passed around, our approach could be described in terms of a "virtual token"-based network. Observe that our *virtual token* will only be passed to nodes which actually want to transmit something, and will never be lost if a node crashes.

In the context of minimalist operating systems, tiny Timber stands out with its deadline-driven scheduling and the heritage of the reactive object paradigm of Timber. This is achieved while matching the resource requirements and performance metrics of popular kernels such as TinyOS [17], Contiki [18], FreeRTOS [19] and AmbientRT [20].

VII. Conclusions and future work

We have examined the possibility of using a FIFO-based protocol for wireless control. Based on the simulations and theoretical results, we conclude that this MAC has suitable timing properties, such as predictable max delays and controllable jitter.

The implementation also demonstrates the relative simplicity of implementing the FIFO WiDOM MAC on an embedded platform, and measurements show that it is viable for medium term deployment on battery powered nodes. The Timber programming model has been demonstrated to efficiently capture the reactive behavior of the system, and a complete protocol implementation has been presented.

We have not covered fault-tolerance or the effects of errors in general but these issues are largely orthogonal to the time related properties. To what extent this holds true, and how existing fault-tolerance methods could be integrated, should be investigated.

In the future, a larger case study involving a real-world control application would be of interest, to examine how the MAC protocol would work in a more realistic setting.

References

- [1] H. Kopetz and R. Obermaisser, "Temporal composability," *Computing and Control Engineering Journal*, vol. 13, no. 4, pp. 156–162, 2002. [Online]. Available: <http://ieeexplore.ieee.org/iel5/2218/22129/01029796.pdf>
- [2] V. Leijon, "FIFO networking: Punctual event-triggered networking," 2006, **Unpublished manuscript**: <http://www.csee.ltu.se/~leijon/publications/fifo-06.pdf>.
- [3] N. Pereira, B. Andersson, and E. Tovar, "Implementation of a dominance protocol for wireless medium access," in *RTCSA '06: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 162–172.
- [4] ISO, *11898 Road Vehicle - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication*. CENELEC, 2003.
- [5] A. Albert, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," in *Embedded World*, 2004, pp. 235 – 252.
- [6] K. Tindell, A. Burns, and A. Wellings, "Calculating controller area network (CAN) message response times," Feb. 16 1995. [Online]. Available: http://citeseer.ist.psu.edu/291115.html;http://www.cis.ksu.edu/~neilsen/classes/cis721/papers/DCCS_TBW_94.ps
- [7] A. Black, M. Carlsson, M. Jones, R. Kiebert, and J. Nordlander, "Timber: A programming language for real-time embedded systems," Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.
- [8] P. Lindgren, J. Nordlander, and J. Eriksson, "Robust Real-Time Applications in Timber," in *Sixth IEEE International Conference on Electro, Information Tech, EIT*, 2006.
- [9] J. Mosely, "An efficient contention resolution algorithm for multiple access channels," Master's thesis, Massachusetts Institute of Technology, May, 1979.
- [10] J. Capetanakis, "The multiple access broadcast channel: Protocol and capacity considerations," Ph.D. dissertation, Massachusetts Institute of Technology, 1977.
- [11] R. Bril, J. Lukkien, R. Davis, and A. Burns, "Message response time analysis for ideal controller area network (CAN) refuted," in *the 5th International Workshop on Real-Time Networks (RTN 06)*, July 2000.
- [12] S. Poledna and G. Kroiss, "The time-triggered communication protocol TTP/C," *Real-Time Magazine*, vol. 4, no. 98, pp. 100–102, 1998.
- [13] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112 – 126, Jan. 2003.
- [14] H. Kopetz, "A comparison of CAN and TTP," in *15th IFAC Workshop on Distributed Computer Control Systems*, 1998. [Online]. Available: citeseer.ist.psu.edu/kopetz98comparison.html
- [15] L. Almeida, P. Pedreiras, and J. Fonseca, "The FTT-CAN protocol: why and how," *Industrial Electronics, IEEE Transactions on*, vol. 49, no. 6, pp. 1189–1201, December 2002.
- [16] N. Malcolm and W. Zhao, "The timed-token protocol for real-time communications," *Computer*, vol. 27, no. 1, pp. 35–41, 1994.
- [17] "TinyOS Community Forum — An open-source OS for the networked sensor regime." <http://www.tinyos.net/>, 2006.
- [18] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *First IEEE Workshop on Embedded Networked Sensors*, 2004.
- [19] "FreeRTOS-A Free RTOS for ARM7,ARM9,Cortex-M3,MSP430,MicroBlaze,AVR,x86,PIC18,H8S,HCS12 and 8051," <http://www.freertos.org/>, 2006.
- [20] "Ambient Systems - for low cost, low power, wireless mesh networking solutions," <http://www.ambient-systems.net/>, 2006.