Response Time Analysis of Composable Micro-Protocols*

João Ventura
Universidade de Lisboa
jcv@ieee.org

João Rodrigues
INETI
joao.carlos@mail.ineti.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

Abstract

The paper presents a generic framework to analyse the timing behavior of protocol graphs derived from the composition of micro-protocols. The model assumes that a protocol stack is composed of a set of protocol objects that interact through the exchange of events. A specific task is associated with each relevant protocol event and for each task, the periods and offsets are derived from a description of the interactions between adjacent protocols. To illustrate the use of the model, a stack of modular reliable group communication protocols for the CAN field-bus is analysed.

1 Introduction

With the increase of processing power and network bandwidth it is possible to build sophisticated distributed hard-real time systems. Many of these systems benefit from communication services that enforce strong consistency properties such as ordering and agreement at the communication level. The construction of such communication systems using the composition of several micro-protocol objects is an approach that has been applied with success in the non real-time arena [2, 1, 6]. This encourages the reuse of protocol components and allows the applications to configure stacks tailored to their needs. To benefit from this approach in hard real-time systems, one must be able to derive the timing behavior of a protocol composition given a description of its protocol objects.

This paper presents a general framework to analyse the timing behavior of protocol graphs derived from the composition of micro-protocols. Individual micro-protocols are described as protocol objects that subscribe and produce events; interactions among adjacent protocols are modeled by the exchange of these events. The protocol implementation is modeled by a set of tasks, each programmed to handle a specific protocol event. The periods and offsets of these tasks can be derived from the protocol interfaces and a description of the traffic load. The model presented is generic but it is being developed to be applied to a concrete protocol composition framework, the *RT-Appia* system [7].

To illustrate the use of the framework, the paper presents the study on the timing analysis of a set of modular fault-tolerant group communication protocols designed for the CAN field-bus: RELCAN over EDCAN [9]. In order to perform this study, an existing software tool was extended to comply with the model requirements.

The paper is organised as follows: Section 2 presents the notion of protocol composition in which the objective of being able to deduce timing analysis of a generic protocol is based. Section 3 presents the theoretical model used in producing the analysis. Section 4 presents the timing analysis of a case-study protocol stack that was developed to provide fault-tolerance communication over the CAN field-bus. Section 5 concludes the paper.

2 Real-Time OO Protocols Stacks

The growing requirements of distributed hard real-time systems demand more diverse and complex communication protocols. Monolithic implementations of communications protocols have many disadvantages. They are hard to expand or refine and may introduce run-time overheads due to functionalities that are not strictly required by the application. Additionally, the timing behavior of monolithic implementations may be difficult to assess.

A successful approach to avoid the pitfalls of monolithic communication systems is to rely on the composition of micro-protocol objects. This approach promotes the reuse of micro-protocols and the construction of protocol stacks that exactly match the application requirements. It may also simplify the timing analysis of the composite stack. To achieve such analysis one needs a description of the timing properties of each micro-protocol and a description of the interaction among protocols in a given stack.

The x-Kernel [4] is an early and influential work on protocol composition. A version of x-Kernel adapted to

*This work has been partially supported by the PRAXIS/P/EEI/14187/1998 project, DEAR-COTS. J. Ventura has been supported by the PRAXIS XXI Programme under grant PRAXIS XXI/BM/20729/99.

real-time operation has been developed in the scope of the CORDS project [12]. Following the initial work with x-Kernel, many other protocol kernels have been designed with enriched functionality. Notable examples are Ensemble [2], RTCactus [3] and *RT-Appia* [7].

In this paper we are interested in designing a framework to extract the timing behavior of compositions of protocols. Even though we are interested in deriving a general framework, we will use the concrete example of the *RT-Appia* protocol kernel.

In *RT-Appia* each stack is composed of one or more *channels*. Each channel is an ordered sequence of *sessions*, instances of a specific *protocol layer*. The session maintains state that is used by the layer to process events. A layer that implements an ordering protocol may maintain a sequence number or a vector clock as part of the session state. The sequence of layers associated with a given channel defines the quality of service implemented by the channel.

Communication between layers is made by exchange of *events*. New events can be created by deriving from a previously defined event class. Each layer declares the set of events the layer produces and that the layer is interested in subscribing. This information allows to identify the relevant interactions among layers.

For each subscribed event, the protocol must provide a handler to process that event. Each handler can be modeled as a real-time task, whose computation time must be known. Handlers for a given layer that share the same session should obtain exclusive access to the session state.

It should be noted that the handling of an event by a given protocol layer may generate multiple events that need to be propagated in the stack. For instance a transport protocol may have to fragment a message into several packets. Thus, the issuing of a single send request event at the transport layer may generate multiple send events at the network layer. This makes the analysis of the timing behavior of a protocol composition a non-trivial task.

In this paper we propose an off-line technique suitable for event-based protocol compositions. To perform the schedulability analysis it is necessary to capture the chain of events with the longest computation time. Note that a chain of events must have a finite number of events. A chain of events terminates when all the sessions return to the idle state. The Worst Case Response Time (WCRT) of a channel activation is derived from the computation time of the worst case chain of that channel and from the WCRT of other higher priority channels.

3 Profiling of protocol compositions

The profiling system must receive as input an abstract description of the protocol composition. The exact form used to supply the description (such as source code or state diagrams) is not relevant for this work, as long as the following information can be extracted:

- The set of micro-protocols and the order by which they process the events.
- The set of events processed by each layer and the set of events that may be generated by the layer during the processing of each event.
- The priorities of these events (these will be directly assigned to the respective handler tasks).

In addition, one needs also to obtain a description of the environment where the micro-protocol composition will execute and the input load. Namely, one needs to know:

- The period of each event received at the stack borders (user requests).
- The worst case computation time (WCCT) for handling each of these events by every interested layer.
- The scheduling parameters for the scheduler that activates the tasks.
- The number of processes communicating (for group communication protocols) and the period of user requests at the remote nodes (the profiling technique derives automatically the worst case load induced by control messages).
- The background load of tasks and messages.
- The communications network scheduling model and parameters.

Using this information, the profiling technique extracts the relevant information required to derive the worst case response time of the protocol composition like the event graph and the task offsets.

Event graphs A fundamental step of the profiling technique consists in extracting the relevant protocol *event graphs*. An event graph is a graph of causally related protocol events whose root is an event triggered at the protocol user interface, such as a user request. The graph is constructed automatically from the information provided to the system as follows: the user request event is delivered to the top layer of the protocol stack. For that event, the set of generated events are added as leafs directly connected to the root. Then, each of these events is delivered to the next layer that registered as their subscribers in the protocol stack. The procedure is executed recursively for each of those events until the corresponding layer on the destination stack delivers the event to its upper layer and no low level events exist.

It should be noted that the protocol graph can, and usually does, span multiple nodes in the system: when the lower protocol layer generates an event associated with the transmission of a network message, a reception event is delivered to the protocol stack of the remote node(s).

Set of protocol tasks A dedicated task is associated to the handling of each event by each protocol layer. The period of these tasks can be extracted automatically from the event graph, and is equal to the period of the root event (the user-supplied load). The WCCT of each task is supplied as an input parameter to the profiling system.

Task Offsets Determination After decomposing the protocol into several tasks it is necessary to assure that the tasks will be scheduled in time to handle event reception without causing unnecessary delays to the protocol. This is done by scheduling them according to a best-case scenario. Each task is scheduled so that incoming events arrive at the same time as the task activation.

In determining the task offsets, it is necessary to assume a best-case scenario, where there are no transmission errors, and the message is the smallest possible. The best-case computation times (BCCT) of the sender task (and its callers) should be used, but since the difference between this and the worst-case time in micro-protocols is very small, and in order to avoid having to supply this extra set of data, the worst-case is still used in this paper. A possible optimisation of the system would also use the best-case times.

Deadlines Verification One of the parameters given by the user for each protocol (event) should be the deadline. The system will verify if this deadline is met, and even if not, it can be used to adjust it in order to be compliant.

Intermediate events should also have a defined deadline, as this is required for recursive steps in the analysis technique, but it is not feasible to assign each of them its own deadline. A feasible solution is to assign one only for the endpoint delivery event, and reuse this value for all the intermediate events. It is simple to observe that either all of them meet this deadline or the endpoint delivery event misses it (a fixed priority scheduler is assumed).

The main component of the system is the timing analysis component which derives all the worst case response times and the worst case message response time.

Timing Analysis Technique The WCRT for each received event of the protocol is computed according to a set timing analysis equations developed by Tindell in [10]. In his work, several schedulability analysis models are presented; we chose to use the timing offset model as this is the one that better allows us to capture the chain of events mechanism. This method assumes that all tasks are periodic, according to the period of the *transaction* to which they belong. A *transaction* is composed by a set of tasks that execute with given offsets in relation to its initial time. We use this to offset the various tasks that make up a micro-protocol in relation to the user request event.

The WCRT of an event is the time taken by the protocol starting at the time of reception of the event that triggered the protocol until it returns to an idle state. This time is composed of the local WCCT and the WCRT of the lower level layers (including lower levels computation time and message transmission through the communication channel).

The timing analysis of the communications protocols is performed in two phases. In the first phase, the WCRT of the processing spent by the protocols themselves is computed. In the second phase, the time spent during messages transmission through the communications network must also be calculated. However, the results of the second phase depend on and affect the results of the first phase, requiring several iterations before converging on a final value.

For the case-study to be presented in the next section, the CAN field-bus model described in [11] is used, but allowing for the interference of the other transactions.

Application to RT-Appia While in our model we consider the existence of a different task to handle each event, in a concrete implementation such as in *RT-Appia*, all the events of a specific layer may be handled by the same operating system task. This does not invalidate the model. On the contrary, by considering different tasks we are modelling the different "personalities" of a single *RT-Appia* task to achieve a finer grain analysis of the timing behavior. It should be noted that the time taken by the *RT-Appia* task to schedule the event will also need to be taken into account when performing analysis of *RT-Appia* protocol stacks.

4 Case-study

To illustrate our profiling technique we have selected a reliable multicast protocol for real-time applications. The protocol, RELCAN, is targeted to the CAN [5] field-bus and it is an interesting target of our analysis since it has been designed as a modular composition of micro-protocols.

4.1 Overview of RELCAN and EDCAN

RELCAN and EDCAN are a set of micro-protocols that provide fault-tolerance capabilities to the CAN field bus. At first look, the CAN specification seems to provide a totally ordered atomic broadcast service. However, as demonstrated by Rufino et al. [9], this is not so. The problem lies in the treatment of an erroneous bit in last bit of the End Of Frame (EOF) delimiter (7 recessive (=1) bits). Since this

```
Sender
ES1 1   when edcan.req(mid⟨type,p,n⟩, mess) invoked at p do
    2       if mess = NULL then
    3           can-rtr.req(mid);
    4       else
    5           can-data.req(mid, mess);
    6   od;

ES2 7   when can-rtr.cnf(mid)
        or can-data.cnf(mid, mess) confirmed do
    8       deliver edcan.cnf (mid,mess);
    9   od;

Recipient
Initialization
init 0  ndup(mid) := 0; // number of duplicates, kept for each message

ER  1   when can-data.ind(mid, mess) received at q
        or can-rtr.ind(mid, mess=NULL) received at q do
    2       ndup(mid) := ndup(mid) + 1;
    3       if ndup(mid)= 1 then // new message
    4           edcan.ind (mid, mess);
    5           if mess = NULL then
    6               can-rtr.req(mid); // clustered
    7           else
    8               can-data.req(mid, mess);
    9           fi;
    10      elif ndup(mid) > j then
    11          can-abort.req(mid);
    12      fi;
    13  od;
```

Figure 1. EDCAN

is the last bit, there is no way to signal the sender that an error was detected, and all nodes that detect the error are forced to accept the frame as correct, to assure a consistent state by all the nodes in the bus. However, suppose that a dominant (=0) bit is detected in penultimate bit of the EOF field by some nodes. These nodes will then begin transmitting an error flag, consisting of 6 dominant bits, beginning at the position of the last bit of the EOF field. If the sender was one of the nodes that had not yet detected an error, it will detect it now and schedule the frame for retransmission. All the other nodes that now detect an error will proceed as explained previously and accept the frame. At this point, some of the nodes have rejected the frame, the sender has scheduled it for retransmission, and the rest have accepted it. If the sender now fails before being able to retransmit the message, then the system is left in a inconsistent state.

To overcome this problem, Rufino et al. proposed in the same paper a protocol that guarantees that if the sender fails, the message is still received by all the other nodes.

EDCAN (see Fig. 1) works by what is called "Eager diffusion". The sender transmits the message on the CAN bus using the generic CAN services and waits for a notification from the controller, delivering then the confirmation corresponding to the request to transmit.

The major changes to the CAN generic services are in the receiver. When a message is received, if it is the first time, it is delivered to the upper layers, and also scheduled for retransmission in the CAN bus. Subsequent duplicates

```
Sender
init 0  rel_sn := 0; // local sequence number

RS1 1   when relcan.req(mess) invoked at p do
    2       rel_sn := rel_sn + 1;
    3       can-data.req(mid⟨R-DATA,p,rel_sn⟩, mess);
    4   od;

RS2 5   when can-data.cnf(mid⟨R-DATA,p,rel_sn⟩, mess) received do
    6       can-rtr.req (mid⟨CONFIRM,p,rel_sn⟩);
    7       relcan.cnf (mess);
    8   od;

Recipient
init 0  ndup(mid) := 0; // number of duplicates, kept for each message
    1   data(mid) := NULL; // data part of the message

RR1 1   when can-data.ind(mid⟨R-DATA,p,n⟩, mess) received at q do
    2       ndup(mid) := ndup(mid) + 1;
    3       data(mid) := mess;
    4       start alarm (mid);
    5       if ndup(mid)= 1 then // new message
    6           relcan.ind (mess);
    7       fi;
    8   od;

RR2 9   when can-rtr.ind(mid⟨CONFIRM,s,n⟩) received at q do
    10      data(mid) := NULL;
    11      cancel alarm(mid);
    12  od;

RR3 13  when alarm(mid) expires at q do
    14          edcan.req (mid, data(mid));
    15  od;

RR4 16  when edcan.ind(mid⟨R-DATA,p,n⟩, mess) received at q do
    17      ndup(mid) := ndup(mid) + 1;
    18      if ndup(mid)= 1 then // new message
    19          relcan.ind (mess);
    20      fi;
    21  od;
```

Figure 2. RELCAN

of the message are ignored, except for keeping track of the number of duplicates. If a maximum of duplicates (j) are received, an abort request is made to abort the pending retransmission, if possible.

EDCAN takes advantage of the possibility of simultaneous transmission of Remote Transmission Request (RTR) frames, for messages with null data fields. This feature is of special interest in the retransmission phase of the protocol, in which several nodes will retransmit the same message, and the probability of simultaneous transmissions is high, thus saving some transmission time on the bus.

As can easily be seen, this protocol assures that if a message is received by at least one non-faulty node, then all the other non-faulty nodes will also receive the message, even if the sender fails. However, this introduces a high overhead on the bus. To alleviate this overhead, Rufino et al. developed RELCAN.

The RELCAN protocol (Fig. 2) assures a reliable communication service, but with less transmission time overhead in the best case than EDCAN. The sender uses a two phase protocol, the first phase consists of putting the mes-

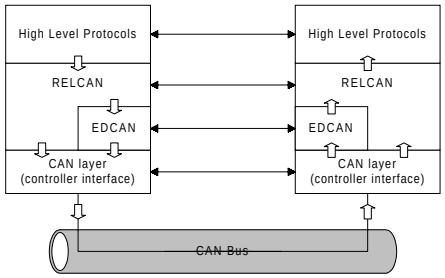

Figure 3. Protocol Stack of RELCAN

sage on the CAN bus, after which it waits for the confirmation of correct transmission from the CAN controller. The second phase consists of sending a CONFIRM message signalling that no retransmissions are required.

The receiver delivers the message to the upper layers when receiving the message for the first time, saves a copy for possible retransmission and starts a timer alarm. If the CONFIRM message is received before the alarm expires, then the alarm is canceled. If the alarm expires, then it means that the sender has failed, and the receivers should retransmit the message using the EDCAN services.

4.2 Analysis of RELCAN (and EDCAN)

4.2.1 Set of micro-protocols

The protocol stack of RELCAN, illustrated in Fig. 3, is composed of RELCAN, EDCAN and CAN itself. CAN is required by both RELCAN and EDCAN, and EDCAN is required by RELCAN.

4.2.2 Events processed and generated

In the following text, the events are referenced using only the event name, as this does not introduce ambiguity. If it did, then the events would have to be referenced using also the event type (message identifiers, etc).

RELCAN The RELCAN sender (Fig. 2) can generate two events for the CAN layer (CAN-DATA.REQ and CAN-RTR.REQ) and one for the upper layer (RELCAN.CNF), and handles one event from the upper layers (RELCAN.REQ) and one from the CAN layer (CAN-DATA.CNF).

The recipient (Fig. 2) can generate one event for the upper layer (RELCAN.IND), one EDCAN event (EDCAN.REQ) and two alarm event (ALARM.START and ALARM.CANCEL). It can handle two CAN layer events (CAN-DATA.IND and CAN-RTR.IND), one alarm event (ALARM.TIMEOUT) and an EDCAN event (EDCAN.IND). The alarm events are assumed to be delivered and generated by an external service.

EDCAN The EDCAN sender (Fig. 1) can generate one upper layer event (EDCAN.CNF) and two CAN layer events (CAN-DATA.REQ and CAN-RTR.REQ), it handles one upper layer event (EDCAN.REQ) and two CAN layer events (CAN-DATA.CNF and CAN-RTR.CNF).

The Recipient (Fig. 1) generates one upper layer event (EDCAN.IND), and three CAN layer events (CAN-DATA.REQ, CAN-RTR.REQ and CAN-ABORT.REQ). It handles two CAN layer events (CAN-DATA.IND and CAN-RTR.IND).

4.2.3 Set of protocol tasks

RELCAN The RELCAN protocol uses two tasks on the sender, and four tasks on the recipient. The first sender task (RS1 on Fig. 2) handles the RELCAN.REQ event and sends the message in a CAN data frame, and the second (RS2) handles the CAN-DATA.CNF event and sends the CONFIRM message.

The recipient tasks are:

- (RR1) The first recipient task handles the reception of the message from the CAN layer (CAN-DATA.IND event) and delivers it.
- (RR2) Handles the reception of the CONFIRM message (CAN-RTR.IND event).
- (RR3) Activated by the ALARM.TIMEOUT event, it begins the retransmission of the message using EDCAN.
- (RR4) Handles the reception of the message from the EDCAN layer (EDCAN.IND event).

EDCAN The EDCAN protocol is implemented by three tasks on the sender, and two tasks on the recipient. Some of these tasks are identical, so they appear only once in Fig. 1, namely the ES2a and ES2b which appear only as ES2, and ERa and ERb which appear only as ER.

The first sender task (ES1) handles the EDCAN.REQ event and sends the message to the CAN layer, it proceeds to wait for confirmation of whether the transmission was successful, the second and third tasks are identical, ES2a handles CAN-DATA.CNF, ES2b handles the CAN-RTR.CNF event, after which they deliver confirmation to the upper layer. The recipient tasks (ERa and ERb) handle the reception of the message (CAN-DATA.IND and CAN-RTR.IND events respectively) and begin retransmission on the first reception.

4.2.4 Worst case computation time

The WCCT depends on the specific architecture used to implement the protocol. Since the actual time is not relevant for the purposes of this study, a uniform time of $150\mu s$ was

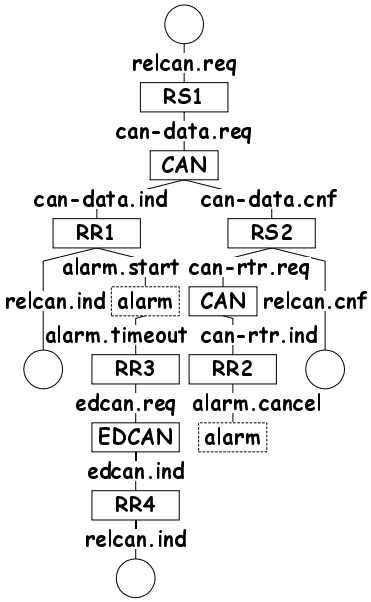

Figure 4. Event graph of RELCAN

chosen for all tasks. Note that with this value for the task execution time it is possible to cause overrun errors during message bursts. If overruns need to be avoided, a smaller execution time must be achieved (using specialized hardware).

4.2.5 Event graphs

Besides the list of handled and generated events it is necessary to link them causally. This is done by processing the micro-protocol algorithm to generate the event graph.

RELCAN The event graph of RELCAN is depicted in Fig. 4. It can be seen both the event chain of the best case scenario (the CONFIRM message is sent and the alarm is canceled) and of the worst case scenario (the CONFIRM message is not sent, and the alarm is triggered, resulting in a call to the EDCAN layer). The worst case response time of these scenarios is studied in the following subsection.

EDCAN In Fig. 5 is a representation of EDCAN's event graph. As expected, it has a symmetry based on the message being empty or not. The recursive character of the protocol is illustrated with the fact that the EDCAN recipient can generate an event that is handled by itself. The dotted lines are events generated by the EDCAN recipient that the simple event graph building technique includes, but that a context-aware examination of the algorithm excludes.

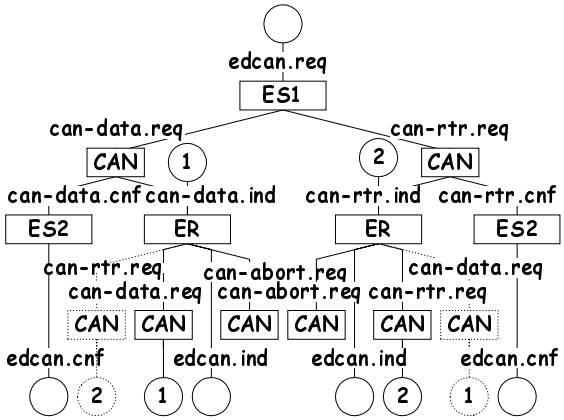

Figure 5. Event graph of EDCAN

4.2.6 Worst Case Response Times

To illustrate the use of the proposed technique, we introduce a very simple scenario that does not involve any interference, jitter and network errors, so that all calculations can be performed manually. The WCRT for a single message to be transmitted using RELCAN is given by the case where the sender fails after sending the data message, but before transmitting the CONFIRM message. This forces the activation of the RELCAN layer. The following equation gives this value:

$$\begin{aligned} &C_{RS1} + C_m + C_{RR1} + alarm + C_{RR3} + \\ &C_{ES1} + C_m + C_{ER} + \max(C_{RR4}, C_m) + \\ &(j-2)\max(C_m, C_{ER}) + C_{ER} \end{aligned} \quad (1)$$

where C_i is the WCCT time of task i, C_m is the worst-case transmission time of message m, $alarm$ is the alarm timeout value and j is the number of recipient nodes on the bus. This chain of events begins with the RELCAN sender sending the message, which is received by the recipient which sets the alarm. The alarm timeout activates the RELCAN recipient task that calls the EDCAN sender task, which begins retransmission of the message. The other nodes will then receive and retransmit the message.

According to [8], one CAN 2.0B message can take a maximum of $157\mu s$ to transmit, in a 1Mbps bus. Using this and the above value for the WCCT, ($C_* = 150\mu s$), and the alarm set to ($400\mu s$), equation 1 is equal to $1928\mu s$, with $j = 3$ nodes on the bus.

Fig. 6 is an illustration of these values: Node 1, the sender, calls RELCAN to send a message, which is correctly received only by node 4, and then fails. Node 4, will process the message and deliver it during execution of its RR1 task, setting the alarm. This will expire after $alarm$ time has passed, executing RR3 in node 4, followed by the ES1 task, which will begin retransmitting the message. This

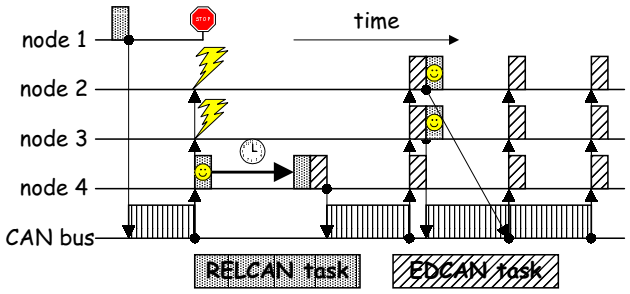

Figure 6. RELCAN in a 4 node bus.

message will be received at all nodes, which will deliver it and one of which will begin retransmission at once, after which the retransmissions of the others will follow until there are no more retransmissions.

Comparing this with the best case (no errors), even with the largest CAN message we have:

$$C_{RS1} + C_{data} + C_{RS2} + \max(C_{rtr}, C_{RR1}) + C_{RR2} \quad (2)$$

which is equal to $757\mu s$, clearly a better value than before. In the best case, the RELCAN protocol does not execute any retransmission, so its response time is equal to the sum of the WCCT taken by the intervening tasks (RS1, RS2, RR2 and RR1; RS2 being of higher priority than RR1), and the transmission time of the messages (one data and one RTR message). Since the processing of RR1 and the transmission of the RTR message are concurrent, only the largest of this value is used (the maximum transmission time of an RTR message is $64\mu s$ [8]).

4.2.7 Task Offsets Determination

The values given in the following paragraphs use equations 1 and 2 and the values presented previously, but using only the parcels before the first activation of the task.

RELCAN The best-case scenario described above can be used to determine the task offsets for the RELCAN tasks. We assume a task offset of 0 for the RS1. RS2 and the RR1 should have the same offset, as they start executing at the end of the first message transmission, equal to $214\mu s$. RR2 should have an offset of $428\mu s$. RR3 should have an offset of $764\mu s$, and RR4 an offset of $1278\mu s$.

EDCAN EDCAN's best case scenario is one where ES1 executes with an assumed offset of $0\mu s$, sends a remote frame message, and all the recipient tasks receive it with an offset of $214\mu s$. ES2 will also have this offset. All the retransmissions will be simultaneous, and the recipients process it with an offset of $428\mu s$.

4.2.8 Deadlines Verification

The deadline for a CAN message is very dependent on its periodicity, as it will be overwritten by the next message if not delivered by then. Using a similar scenario to the above, it is possible to say that the period of the task invoking the RELCAN protocol must be larger than $664\mu s$ (smallest message with no errors) or it will fail to meet the deadline. Allowing the sender to fail (a hot-spare configuration?), the deadline must be greater than $1728\mu s$ (smallest message and maximum RTR overlap).

4.3 A more complex example task set

The analysis performed above does not take into account scheduler overheads, and interference from other tasks. In order to get a real view of the analysis technique and of the behavior of the RELCAN protocol, a more complex scenario is required. Simple results, like the ones obtained previously can be feasibly obtained manually, but the complexity of the calculations with larger scenarios require the use of an automated tool. With that goal in mind, we extended the software tool that Tindell used in [10] to include communication networks (in this case CAN) and multiple processors. This allowed us to validate the previous results and generate the values in Table 1.

The developed scenario consists of three transactions running on 6 nodes. Each transaction corresponds to the chain of events generated by a user request to the RELCAN layer. All the tasks (and messages) in transaction 1 have higher priority than the ones in transaction 2, which in turn has higher priority than the ones in transaction 3. The following values were used: $T_{clk} = 1$, $C_{clk} = 0$, $C_{QL} = 0$, $C_{QS} = 0$ (no scheduler overheads), the transaction period $T_{ti} = 3000$ for all transactions and $e_i = 1$ (the tasks are activated in every transaction), $D_i = T_{ti=trans(i)}$ (deadline equal to the period of the transaction), and no jitter ($J_i = 0$) for all the tasks. Note that the same offset for tasks in different transactions does not mean that they will start at the same time. The offset is in relation to the start of the transaction, which can occur at any time during the execution of the other transactions.

Comparing these results, it's easy to see that even for the highest priority transaction, the interference from other tasks can be noted, since the CAN bus does not preempt lower priority messages, and so the higher priority message from transaction 1 is delayed $156\mu s$. If comparing these times with the ones obtained earlier for a transaction executing alone ($757\mu s$), transaction 1 takes 130%($983\mu s$), transaction 2 279%($2111\mu s$) and transaction 3 426%($3226\mu s$) more. We believe that the WCRT of RELCAN grows linearly with the number of concurrent transactions. Also note that, although transaction 3 misses the deadline, using $D = 3000\mu s$, the obtained values remain the same for

	transaction 1				transaction 2				transaction 3			
	task	*cpu*	*offset*	*WCRT*	*task*	*cpu*	*offset*	*WCRT*	*task*	*cpu*	*offset*	*WCRT*
high	RS1	1	0	150	RS1	2	0	300	RS1	3	0	600
p	RS2	1	463	150	RS2	2	834	300	RS2	3	1198	600
r	RR1	1	463	300	RR1	1	834	600	RR1	1	1198	1050
i	RR1	2	463	150	RR1	2	834	600	RR1	2	1198	900
o	RR1	3	463	150	RR1	3	834	300	RR1	3	1198	900
r	RR1	>3	463	150	RR1	>3	834	300	RR1	>3	1198	600
i	RR2	1	833	150	RR2	1	1511	600	RR2	1	2176	1050
t	RR2	2	833	150	RR2	2	1511	300	RR2	2	2176	900
y	RR2	3	833	150	RR2	3	1511	300	RR2	3	2176	600
low	RR2	>3	833	150	RR2	>3	1511	300	RR2	>3	2176	600
	msg		*offset*	*WCRT*	*msg*		*offset*	*WCRT*	*msg*		*offset*	*WCRT*
	R-DATA		150	313	R-DATA		300	547	R-DATA		600	598
	CONFIRM		613	220	CONFIRM		1134	377	CONFIRM		1798	378

Table 1. Complex task set results

$D \geq 2776\mu s$. Nodes number 4,5 and 6 exhibit the same temporal behavior, as they run the same tasks, and do not send messages to the bus. This last result could be used to deduce that RELCAN does not degrade with the number of nodes on the bus, however this is only true if EDCAN is never activated, or if the simultaneous retransmission capabilities are fully used (only possible if the message does not contain any data).

5 Conclusions

In this paper we have proposed a framework to analyse the timing behavior of micro-protocols for use in real-time systems. The proposed technique was applied manually to a simple set of real-time micro-protocols, which allowed us to validate it and also to illustrate its use with a relevant example. The timing values were obtained through a specially adapted software tool. The resulting values provide usefull insights on the performance of the RELCAN protocol. The proposed technique is being considered for inclusion in the set of tools for the *RT-Appia* system, even though the task model is different. Having demonstrated its feasibility, it would be interesting to complete the tool that implements this technique. For instance, at the current time the tool considers that a task, in response to a given event, produces the complete set of generated events. This is not always the case since, depending on the context (message size, etc), only a subset of the generated events can be produced. The tool could be improved to exploit contextual information to extract more accurate event graphs. Finally, with the tool completed it would be interesting to perform this analysis in other more complex protocols.

References

[1] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, Nov. 1998.

[2] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[3] M. A. Hiltunen, R. D. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing qos attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, June 1999.

[4] N. C. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engeneering*, 17(1):64–76, Jan. 1991.

[5] ISO, editor. *ISO International Standard 11898 - Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for high-speed communication*. ISO, nov 1993.

[6] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Internation Workshop on Groupware - CRIWG'99*, pages 338–342, Cancún, México, Sept. 1999. IEEE.

[7] J. Rodrigues, H. Miranda, J. Ventura, and L. Rodrigues. The design of RT-Appia. In *Proceedings of Sixth International Workshop on Object-oriented Real-Time Dependable Systems*, page (to appear), Rome, Italy, Jan. 2001.

[8] J. Rufino. An overview of the controller area network. In *Proceedings of the CiA Forum CAN for Newcomers*, Braga, Portugal, Jan. 1997.

[9] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *Digest of Papers, The 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 150–159, Munich, Germany, June 1998. IEEE.

[10] K. W. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS221, Department of Computer Science, University of York, Jan. 1994.

[11] K. W. Tindell, A. Burns, and A. Wellings. Calculating controller area network (can) message response times. In *Proceedings of the IFAC Workshop on Distributed Computer Control Systems*, Toledo, Spain, Sept. 1994. IFAC.

[12] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the 2nd IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Laguna Beach, CA, Feb. 1996.