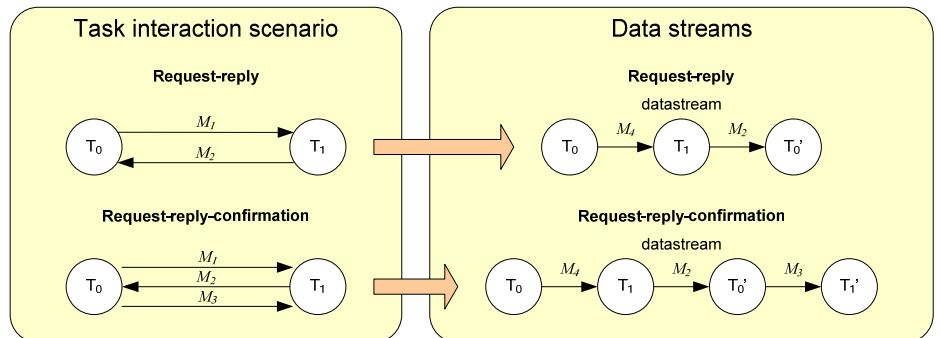**Mário João Barata Calha**

Flexibilização em Sistemas Distribuídos – Uma Perspectiva Holística

A Holistic Approach Towards Flexible Distributed Systems

Mário João Barata Calha

Flexibilização em Sistemas Distribuídos – Uma Perspectiva Holística

A Holistic Approach Towards Flexible Distributed Systems

tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Informática, realizada sob a orientação científica do Dr. José Alberto Gouveia Fonseca, Professor Associado do Departamento de Engenharia Electrónica e Telecomunicações da Universidade de Aveiro

Dedico este trabalho a todos os seres sencientes.

o júri / the jury

presidente / president

Doutor Telmo dos Santos Verdelho
Professor Catedrático da Universidade de Aveiro

Doutora Françoise Simonot-Lion
Professora Catedrática do Institut National Polytechnique de Lorraine – Ecole Nationale Supérieure

des Mines de Nancy

Doutor Urbano José Carreira Nunes
Professor Associado da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Doutor Francisco Manuel Madureira e Castro Vasques de Carvalho
Professor Associado da Faculdade de Engenharia da Universidade do Porto

Doutor Alexandre Manuel Moutela Nunes da Mota
Professor Associado da Universidade de Aveiro

Doutor José Alberto Gouveia Fonseca
Professor Associado da Universidade de Aveiro

Doutor Luís Miguel Pinho de Almeida
Professor Auxiliar da Universidade de Aveiro

agradecimentos

A José Alberto Gouveia Fonseca, Professor da Universidade de Aveiro e meu orientador, a quem expresso os meus sinceros agradecimentos pela supervisão, apoio, incentivo e cordialidade ao longo de todo este trabalho.

A Luís Almeida, da Universidade de Aveiro, pelo encorajamento, disponibilidade e valiosos contributos para este trabalho.
A Paulo Pedreiras, da Universidade de Aveiro, pela colaboração nas fases iniciais deste trabalho.
A Valter Silva, da Universidade de Aveiro, pela disponibilidade e colaboração em diversas fases deste trabalho.
A Joaquim Ferreira, da Escola Superior de Tecnologia de Castelo Branco, pelas muitas vezes que me aproximou da Universidade de Aveiro compensando a distância física que quase sempre existiu.

A todos os colegas e amigos, tanto da Escola Superior de Tecnologia de Castelo Branco como do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro, pela amizade e encorajamento manifestado ao longo destes anos contribuindo significativamente para o desenvolvimento deste trabalho.

E também à Mafalda, à Matilde e à Madalena, pelo encorajamento e pelas muitas vezes que não as pude acompanhar.

palavras-chave   Sistemas distribuídos de tempo-real, análise do fluxo de informação, determinação de parâmetros, simulação holística e núcleos de sistema operativo.

resumo   Em sistemas distribuídos o paradigma utilizado para interacção entre tarefas é a troca de mensagens. Foram propostas várias abordagens que permitem a especificação do fluxo de dados entre tarefas, mas para sistemas de tempo-real é necessário uma definição mais rigorosa destes fluxos de dados. Nomeadamente, tem de ser possível a especificação dos parâmetros das tarefas e das mensagens, e a derivação dos parâmetros não especificados. Uma tal abordagem poderia permitir o escalonamento e despacho automático de tarefas e de mensagens, ou pelo menos, poderia reduzir o número de iterações durante o desenho do sistema. Os fluxos de dados constituem uma abordagem possível ao escalonamento e despacho holístico em sistemas distribuídos de tempo-real, onde são realizadas diferentes tipos de análises que correlacionam os vários parâmetros. Os resultados podem ser utilizados para definir o nível de memória de suporte que é necessário em cada nodo do sistema distribuído.

Em sistemas distribuídos baseados em FTT, é possível implementar um escalonamento holístico centralizado, no qual se consideram as interdependências entre tarefas produtoras/consumidoras e mensagens. O conjunto de restrições que garante a realização do sistema pode ser derivado dos parâmetros das tarefas e das mensagens, tais como os períodos e os tempos de execução/transmissão. Nesta tese, são estudadas duas perspectivas, uma perspectiva centrada na rede, i.e. em que o escalonamento de mensagens é feito antes do escalonamento de tarefas, e outra perspectiva centrada no nodo.

Um mecanismo simples de despacho de tarefas e de mensagens para sistemas distribuídos baseados em CAN é também proposto neste trabalho. Este mecanismo estende o já existente em FTT para despacho de mensagens. O estudo da implementação deste mecanismo nos nodos deu origem à especificação de um núcleo de sistema operativo. Procurou-se que este introduzisse uma sobrecarga mínima de modo a poder ser incluído em nodos de baixo poder computacional.

Neste trabalho, é apresentado um simulador, SimHol, para prever o cumprimento temporal da transmissão de mensagens e da execução das tarefas num sistema distribuído. As entradas para o simulador são os chamados fluxos de dados, que incluem as tarefas produtoras, as mensagens correspondentes e as tarefas que utilizam os dados transmitidos. Utilizando o tempo de execução no pior caso e o tempo de transmissão, o simulador é capaz de verificar se os limites temporais são cumpridos em cada nodo do sistema e na rede.

keywords

abstract

In distributed systems the communication paradigm used for intertask interaction is the message exchange. Several approaches have been proposed that allow the specification of the data flow between tasks, but in real-time systems a more accurate definition of these data flows is mandatory. Namely, the specification of the required tasks' and messages' parameters and the derivation of the unspecified parameters have to be possible. Such an approach could allow an automatic scheduling and dispatching of tasks and messages or, at least, could reduce the number of iterations during the system's design. The data streams present a possible approach to the holistic scheduling and dispatching in real-time distributed systems where different types of analysis that correlate the various parameters are done. The results can be used to define the level of buffering that is required at each node of the distributed system.

In FTT-based distributed systems it is possible to implement a centralized holistic scheduling, taking into consideration the interdependences between producer/consumer tasks and messages. A set of constraints that guarantee the system feasibility can then be derived from tasks and messages' parameters such as the periods and execution/transmission times. In this thesis the net-centric perspective, i.e., the one in which the scheduling of messages is done prior to the scheduling of tasks, and the node-centric perspectives are studied.

A simple mechanism to dispatch tasks and messages for CAN-based distributed systems is also proposed in this work. This mechanism extends the one that exists in the FTT for the dispatching of messages. The study of the implementation of this mechanism in the nodes gave birth to the specification of a kernel. A goal for this kernel was to achieve a low overhead so that it could be included in nodes with low processing power.

In this work a simulator to preview the timeliness of the transmission of messages and of the execution of tasks in a distributed system is presented. The inputs to the simulator are the so-called data streams, which include the producer tasks, the correspondent messages and the tasks that use the transmitted data. Using the worst-case execution time and transmission time, the simulator is able to verify if deadlines are fulfilled in every node of the system and in the network.

Contents

A   List of publications

List of Figures

List of Tables

1 Introduction

1.1 Overview

For a process to be controlled, it is necessary to have clearly defined what information the controller must acquire and what information the controller must produce. If the different elements that constitute the controller are implemented using a distributed approach, then the granularity of the information flow is increased because the data consumption and production may be defined at the level of the computational units. An information flow will now show what information is exchanged between the computational units. Another aspect is the timing. The time between data acquisition, or consumption, and data production might be constrained and in this case, for a system to be correct, it has to produce the right results within a specific time window.

In a distributed embedded system the computational units are allocated to different nodes and each computational unit can process data from different information flows. Ideally, each data flow would have its own path between computational units and in this case there would be no interference between them. This kind of interconnection between nodes is called point-to-point. The disadvantage of the point-to-point connections is that there is an increasing difficulty as the number of data flows where the computational unit participates also increases. Another approach is to have a common path through which all data flows. This has the disadvantage of the possibility of interferences between the data flows but leads to a simpler and more flexible system design.

The Flexible Time-Triggered (FTT) mechanism has been developed to control the interferences between data flows in shared networks, keeping a flexible scheduling and dispatching and respecting timeliness guarantees. It was planned to work just with the messages transmitted in the network. This was considered a limitation and, in consequence, the starting point for the development of this work was the extension of the Flexible Time-Triggered (FTT) mechanism so that tasks could be also considered on the centralized scheduling and dispatching [CF02]. The trigger message of the FTT was adapted in order to accommodate the data fields to convey information about which tasks must be dispatched in the current time slot. From here a basic architecture of a nano-kernel for the nodes was developed [CSFM06]. Following the idea of the FTT, the functionality

of each node does not need to include a scheduler or even a dispatcher, because these units are centralized in some other node. Therefore, the kernel for each node of the system offers only basic functionalities from the task management to the message handling and, in particular, the trigger message handling. With this extension it became possible to use the FTT paradigm as a support for a holistic perspective of a distributed real-time system.

A second part of the work, focused on an issue that was considered relevant to the holistic scheduling that was the parameter determination. The starting scenario was, on one hand, a constrained network where traffic was intense and, on the other hand, more relaxed nodes with low computational load. From this scenario, net-centric approaches were developed [CF04]. These make use of, almost, fully specified messages' parameters in order to derive the remaining messages' parameters and the timing parameters of related tasks. With this contribution it could be possible to have some level of automation in the parameter determination that could assist a system designer.

An option that was taken in the beginning of the study of net-centric approaches was the non-over lapping of the execution and transmission windows of directly related entities. For instance, the directly related entities of a task are its consumed and produced messages, while the directly related entities of a message are its producer task and its consumer tasks. As a simplification, in the first studies it was considered that each task may only consume, or produce, at most, one message. This means that only simple data streams were handled. A technique that was explored in some of these approaches was a coupling between the parameter determination and the scheduler. This was called scheduling dependent parameter determination, while a decoupled approach was called scheduling independent parameter determination.

After these preliminary studies, a simulator, SimHol, was developed [CF03a]. The SimHol implemented the equations that were derived for the various net-centric approaches. With this tool it was possible to test several scenarios, have a practical perspective and fine tune the parameter determination equations.

This part of the work was concluded with the study of the node-centric approaches and the derivation of some equations.

A third part of the work had its emphasis on a generalization of the data streams and their associated issues [CF05]. To start with, other types of task interactions were considered and their representation in the form of graphs was proposed. From here a methodology to

the identification of the data streams was developed. From an implementation point of view, it was found that a suitable data structure was necessary in order to both accommodate these data streams and to support various types of analysis. One of these analyses has to do with the interdependences between data streams, where a task participates in more than one data stream. This has an impact upon the release instant of such task because this parameter has to be suited to all involved data streams. This study led to the possibility of closed loops in the graphs that, when present, make this approach to the parameter determination useless. It was found that this can happen every time a task changes its role. A methodology was proposed to detect these closed loops and to suggest which tasks that are changing their role should be changed. Another analysis to the data streams can determine the necessary level of buffering required to temporarily store messages that are consumed by several tasks. This buffering is defined for each time slot.

Finally, the folding of data streams was explored in order to support the concept of real-time procedures [CSF05]. A real-time procedure has a deadline associated but at the beginning might not be mapped to a set of communicating tasks. When further on the tasks are defined it is necessary to transpose the procedure's deadline specification to the deadline of each task. This aspect was studied and a methodology was proposed.

In between, a parallel extension to this work was studied. This consists on the development of a middleware architecture based on JAVA that would appear with the same functionality to the tasks and messages independently of having a simulator beneath or the real system. With this it would be possible to simulate exactly the same entities that would be used in the real system without any changes.

1.2 The thesis

The thesis supported by the present dissertation argues that:

> *The scheduling of real-time tasks and messages can benefit from approaches that result from a tighter coupling with parameter determination. The FTT paradigm can be used as a basis for the implementation of such a scheduling and also extended so that both tasks and messages can be dispatched.*

1.3 Contributions

The major contributions of this dissertation are summarized next.

1.3.1 Identification of the data streams in a distributed real-time system

An accurate knowledge of the data flows in distributed real-time systems is mandatory in order to better prepare a holistic scheduling. Many works in this area concentrate in finding and using optimal algorithms that can guarantee the system schedulability but studies in parameter determination and tuning are less common. In fact, an identification of the data streams presents itself as an invaluable tool that can assist the parameter determination and tuning of tasks and messages. A technique for this identification and a corresponding representation are proposed. These are based in two main goals which are: encapsulating the computational aspects of tasks and focusing on the interactions between tasks. With this approach, for this study, the most relevant aspects of tasks and messages can be emphasized. This proposal was essayed using the FTT-CAN but it can be used with many other paradigms.

1.3.2 Approaches to the parameter determination of tasks and messages

Several approaches to the parameter determination are presented. These can be integrated in order to create a semi-automated parameter determination and tuning. This support permits considering other system properties like the network and node loads, or the execution and transmission windows. Some approaches can be tightly connected to the scheduler offering some feedback that supports the parameter tuning. The approaches can directly improve aspects like the release jitter of tasks and messages. The approaches can be divided in node-centric and net-centric depending on which resource is more constrained. This proposal was essayed using the FTT-CAN but it can be used with many other paradigms.

1.3.3 Determination of the buffering level for messages in transit

When a message is consumed by more than one task, most probably, it will be consumed by each of them in different time slots. When a message is in transit, i.e. has been produced but has not been consumed, it has to be stored in some buffer. A technique based on the data streams is proposed in order to determine the level of buffering for each node in each time slot. With this, a system designer can more easily define the memory requirements for the kernel of each node.

1.3.4 Real-time procedures that map to several communicating tasks

Many times a system designer has only a deadline for a procedure but cannot easily define the individual deadlines of the corresponding tasks. Also this procedure can be remapped to a different set of tasks or the tasks can be allocated to different nodes. All this has an impact on the tasks' parameters. A proposal for the handling of real-time procedures based on the data streams is proposed.

1.3.5 Task dispatching through the extension of the FTT-CAN

The FTT-CAN protocol [APF02] supports the centralized message dispatching but tasks were not considered. A solution to the task dispatching through an extension of the FTT paradigm is proposed. This solution is based on the extension of the trigger message in order to accommodate information regarding which tasks must be dispatched on the corresponding elementary cycle.

1.3.6 SimHol: a simulation and configuration tool

The simulation of distributed real-time systems plays an important role both for the offline scheduling and for testing online scheduling with scenario changes. A simulator must offer a suitable test bed for this kind of systems providing an accurate, as possible, view of the real working system. The SimHol offers a suitable experimental platform for the simulation of distributed systems based on the time-triggered (FTT) paradigm. Using a simple interface it allows the simulation of various interconnection architectures with different scheduling algorithms. This simulator has validated the set of requirements previously derived, namely the data flow analysis and precedence requirements. Another benefit derived from the design of the SimHol was the definition of an XML format for all input and output data from its modules.

1.4 Organization

In order to support the thesis previously stated, this dissertation is organized as follows:

Chapter 2 – Presents the real-time periodic task and message models used in this thesis. It also addresses the types of scheduling that are commonly used in distributed real-time systems.

Chapter 3 – This chapter focuses on some relevant aspects of time-triggered architectures and communication protocols.

Chapter 4 – Discusses how information flows in typical task interactions considering both unicast and multicast scenarios. From these scenarios, the individual data streams are extracted and restrictions that apply are identified. The possibility of data stream folding is also explored through the concept of real-time procedures. These procedures have precise timing requirements and aggregate various tasks with undefined timing characteristics. Techniques for the parameter determination towards a holistic scheduling are also presented.

Chapter 5 – Presents the implementation aspects of the techniques discussed in the previous chapter and an architectural solution to the dispatching of tasks that is based on an extension of the FTT-CAN.

Chapter 6 – This chapter covers the analysis, design and implementation of a simulator, SimHol, which was developed to test the proposed framework. Two versions of the SimHol are presented, a preliminary version that covered the basic parameter determination and scheduling and a revised version that covers all the main aspects proposed in this thesis. Some experimental results are presented and discussed.

Chapter 7 – A case study is used as an experimental demonstration of the concepts presented in the previous chapters. This case study is based on the CAMBADA robots.

Chapter 8 – Reveals a potential JAVA implementation of the extended FTT-CAN paradigm through the concept of FTTlet.

Chapter 9 – Sets the conclusion of the dissertation and points out several directions for future work.

2 Scheduling tasks and messages

When a single processor has to execute a set of concurrent tasks – that is, tasks that can overlap in time – the CPU has to be assigned to the various tasks according to a predefined criterion, called a scheduling policy [But00]. The set of rules that, at any time, determines the order in which tasks are executed is called a scheduling algorithm.

The notion of priority is commonly used to order the concurrent access to a shared resource. This resource can be a Central Processing Unit (CPU), a network, a bus … . A schedule is a particular assignment of entities to the resource. The order of entities is usually kept in a list called ready queue.

The entities to be scheduled can be very different in nature or function, but they must have some parameters in common. Contention for resources is resolved in favour of the entity with the higher priority that is ready to run. Even if the resource has a capacity that largely exceeds the typical needs, in real-time systems where the critical entities have time constraints, some type of scheduling must be used.

The study of fieldbus based distributed systems, from a perspective that joins both tasks and messages, has already been significantly addressed by the scientific community, either specifically or indirectly when different scheduling scenarios are considered. The subject of schedulability analysis is covered from different points of view in [TC94], [PH98], [PH99], [CP00], [RCR01], [PEP02] and [APF02]. An algorithm for planning the execution of task groups is presented in [BBGT96]. Priority assignment is studied in [RRC01] and [RRC03] which also address task allocation. A more detailed analysis of such works follows.

The importance of pre-run-time scheduling in order to guarantee the timing constraints in large complex hard real-time systems is explored by Xu and Parnas [XP90]. They have examined some of the major concerns in pre-run-time scheduling and considered what formulations of mathematical scheduling problems could be used to address those concerns. In [XP91] they have presented an algorithm that finds an optimal schedule on a single processor for a given set of processes. This algorithm satisfies the deadlines, a given set of precedence relations and a given set of exclusion relations defined on ordered pairs of process segments. This algorithm can be applied to automate the pre-run-time

scheduling of processes with arbitrary precedence and exclusion relations in hard real-time systems.

A first approach to the verification of end-to-end response times for distributed real-time software systems is the holistic scheduling analysis proposed by Tindell and Clark [TC94]. They have shown how to analyze distributed hard real-time systems conforming to a particular architecture – simple fixed priority scheduling of processors, with a simple TDMA protocol arbitrating the access to a shared broadcast bus. The software architecture is a simple one, with periodic and sporadic tasks communicating via messages and shared data areas. They have derived a schedulability analysis for fixed-priority tasks with arbitrary deadlines, and then they have used this analysis to determine the worst-case response times of messages sent between processors. They have extended the processor schedulability analysis to address the delivery costs of messages, both to bound the overhead on a destination processor and the delivery times of the messages, thus obtaining the end-to-end response times. The concept of a 'protected object' was introduced, where a priority-ceiling semaphore is used to guard access to a Hoare monitor. The concurrency control requirements of each task is then characterised by the objects and methods accessed by that task.

Later the same authors have applied the same type of analysis to the bounding of message delays across a communications system. A number of processors are connected to a shared broadcast bus. For a processor to transmit on the bus it must have exclusive access to the bus. They have assumed that the TDMA protocol is used to arbitrate between processors when accessing the bus. Each processor is permitted to have an independent slot size. Messages are assumed to be broken up into packets by the sender task, with large messages requiring several packets. Each message is assigned a fixed priority, and all packets of the message are given this priority. Each message must have a unique destination task, and no task can receive more than one message. These restrictions are needed in order to bound the peak load on the communications bus, and to enable the schedulability of the destination tasks to be determined. They have also shown how the analysis can be integrated to provide a powerful model of distributed hard real-time computation.

The most important aspect of integrating the processor and communications schedulability analysis is to bound the overheads due to packet handling on a given processor. Tindell and Clark have also shown that the holistic scheduling equations cannot be trivially solved. A

solution to this problem can be found by realising that all of the scheduling equations are monotonic in window size, response time and release jitter. Therefore, it is possible a recurrence relation and iterate to a solution setting the inherited release jitter, for the first iteration, for all tasks to zero. One of the restrictions of the computational model is that task access to a protected object must always be local: it is not permitted to lock a semaphore on another processor. One way around this potential problem is to use an approach akin to RPC. Another restriction of the model is that no task can receive more than one message. The real benefit of the analytical approach taken in their work was not just to obtain a priori schedulability guarantees across a distributed system, but to aid the configuration of such a system.

Some real-time applications require that not only single processes be executed within given deadlines, but whole groups of processes be considered as atomic entities: the execution of the whole group is a benefit, while the execution of only some of the components may be useless, thus a waste of resources, or even a loss or a damage. This is the case, for example, of a transaction executed in a distributed environment or of applications with end-to-end constraints. This is also the case of real-time applications with some dependability requirements. Indeed, a typical means to ensure dependability in a system is to embed redundancy in its design substituting a single task with a set thereof. If this is made on an application with real-time constraints, then the constraints of a single task must become constraints of groups of tasks and the system on which the application has to run must guarantee the execution of task groups with these constraints.

Bizzarri *et al* [BBGT96] investigated the problem of planning groups of tasks in real-time environments. They first discuss the issues related to the design choices and their implications on planning strategies for tasks groups. Then an algorithm for planning groups of tasks is proposed for the specific case of fault tolerant real-time applications where fault tolerance is realized by groups of tasks forming together fault tolerant structures. Real-time critical activities impose requirements both on timeliness and on dependability of the executed computations; considering this context, the necessary design choices that allow to precisely characterizing a suitable planner are operated. In order to assure a satisfactory level of dependability, redundancy is utilized to cope with the occurrence of faults.

Palencia & Harbour [PH98] have presented a technique for analysing tasks with static and dynamic offsets in the context of preemptive fixed-priority scheduling. The system is composed of periodic transactions, each containing several tasks. Each task is released after some time, called the offset, elapsed since the arrival of the event that triggers the transaction. This work is less restrictive than the work by Tindell in which the task offsets are static and restricted to being smaller than the tasks' periods, thus just useful in those systems where task activations are timed precisely, at periodic intervals. However, a technique to calculate the worst-case response times of task sets with offsets could be very valuable to obtain a solution to the problems of task suspension in distributed systems. In particular, if task offsets could be dynamic, i.e. if they could change from one activation to the next. For example, in distributed systems a task may be released when a previous task completes its execution and a message is received; this release time can vary from one period to the next.

In distributed systems it is common that task deadlines are larger than the periods, and thus it is also very likely that task offsets might become larger than the task periods. Consequently, in the referred Palencia & Harbour paper, Tindell's analysis of tasks with static offsets is extended in the following ways: through eliminating the restriction of task offsets being smaller than the period; thus providing a formal basis that overcomes some defects in Tindell's work; and most important, extending the technique to cover the case in which task offsets may vary dynamically, thus making the technique directly applicable to the analysis of distributed systems and systems in which task suspend themselves.

In distributed systems the new technique allows a significant increase of the schedulable utilization of the CPU compared to the case when previous analysis techniques were used. This comes at no cost for the application, which will still be scheduled using fixed priorities.

Improved techniques for the schedulability analysis of tasks with precedence relations in multiprocessor and distributed systems have been presented also by Palencia & Harbour [PH99]. These techniques are based on the analysis of tasks with dynamic offsets that they had previously developed, which they have improved in this work by exploiting the precedence relations in a more accurate way, and considering the priority structure of the different tasks.

The system model is composed of a set of tasks executing in the same or different processors, which are grouped into entities called transactions. Each transaction is activated by a periodic sequence of external events (periodic) and contains a set of tasks. Each task has its own unique priority, and the task set is scheduled using a preemptive fixed priority scheduler. When the activation of the task occurs with an offset that is constant, independently of the execution of other tasks in the system, they call it a static offset. An offset is called dynamic if it can vary between some minimum and maximum interval. This variation is often caused by the execution of other tasks or activities for which the activated task must wait.

In the referred work, deadlines are allowed to be larger than the period, and so at each time there may be several activations of the same task pending. It is also allowed both the offset and the jitter to be larger than the period of its transaction. For each task the response time is defined as the difference between its completion time and the instant at which the associated external event arrived. To calculate the worst-case global response time of a task, a worst-case scenario for its execution must be built. The main problem is that the response times are dependent on the task offsets, and the task offsets depend on the response times. The solution to this problem can be found in the WCDO (worst case dynamic offsets) iterative method, based upon Tindell & Clark's holistic analysis. Through simulation results, it was shown that the benefits of the new analysis over the previous analysis techniques for distributed and multiprocessor systems are very high. The response times with the new technique are significantly lower, and the maximum schedulable utilization can be increased. In the examples shown, it was increased by an additional 11% of schedulable utilization.

Chevochot & Puaut [CP00] dealt with a complex run-time support with fault-tolerance capabilities and made of multiple tasks that invoke each other. They have taken in consideration not only the temporal behaviour of the application tasks but also the behaviour of the run-time support in charge of executing the applications. Their paper is devoted to the schedulability analysis of a run-time support for distributed dependable hard real-time applications, through an adaptation of the distributed system arbitrary deadline analysis developed by Tindell for fixed priority scheduling. Two approaches can be used to obtain task response times in distributed real-time systems. On one hand, one can simultaneously consider the application tasks and the run-time support tasks. On the other

hand, the run-time support tasks can be considered separately; the knowledge of the system worst-case load scenario is then used to compute the run-time support tasks response time. In contrast to previous works that consider rather simple systems, their work deals with a complex run-time support that incorporates fault-tolerance mechanisms, and which is made of multiple tasks that invoke each other.

Richard et al [RCR01] have presented a method to handle complex asynchronous communication relations between tasks in a hard real-time distributed system in order to prove its schedulability using the holistic analysis. The method is based on the unfolding of the generalized precedence graph underlying to the complex communication relations. A new set of tasks and a new set of messages are then created such that all dependent tasks have the same period. This new task set is shown to be equivalent from the schedulability point of view and can be directly used to validate the application with the classical holistic analysis. Since their method only focus on the precedence relations among the tasks and the messages, they made no assumption on the architecture of the hard real-time distributed system, on the scheduling policies of the processors or the network, on the concurrency control protocol and also on the synchronization protocol of the messages. All these parameters of the distributed system are managed in the holistic analysis that uses in entry the problem defined by the proposed algorithm. The method can be applied on single processor problems as well as on complex distributed systems. In some particular cases, the generalized precedence graph unfolding is not necessary. For instance if a generalized precedence constraint between two tasks mapped on the same processor and having proportional periods is considered, then one can enforce the scheduling policy to obey to the precedence relation. Such result can be achieved by modification of the task parameters without creating any duplicate of the initial task. So a perspective of this work is to search new conditions in order to avoid generalized precedence graph unfolding.

An optimal priority assignment for fixed-priority tasks and messages in an automotive computerized system has been presented also by Richard *et al* [RRC01]. The architecture is based on multiple fieldbus networks connecting uniprocessor computation units. Tasks and messages are on-line scheduled according to fixed priorities. The priority assignment is performed with a branch and bound algorithm. Goal vertices are proved schedulable or not using the holistic analysis. The solution space is modeled with a tree explored using a depth-first search strategy. For every explored vertex, lower bounds of worst-case response

times are computed. These lower bounds are used to prune unfeasible vertices. The lower bound scheme is based on the adaptation of the holistic analysis for tasks without a priori known priorities.

After, Richard *et al* [RRC03] have presented a Branch and Bound method that automatically allocates tasks to processors and assigns fixed priorities to tasks. They have shown how to simultaneously allocate tasks and assign their priorities and to use the principles of the holistic analysis to calculate lower bounds of worst-case response times for tasks and messages. Numerical experimentations have shown that the method can find feasible schedules even if the workload of the system is high. The method is applicable for real-size applications. They have used an example with two networks, three pools of processors that include 9 processors, 44 tasks and 19 messages.

Pop *et al* have presented in [PEP02] and [PEP03] a holistic scheduling and timing analysis approach for applications consisting of both event-triggered (ET) and time-triggered (TT) tasks. A static cyclic schedule is constructed for TT tasks and static messages and the schedulability of ET tasks and dynamic messages is verified. The static schedule is constructed in such a way that it fits the schedulability requirements of the ET domain. They have considered a bus access optimization problem and have shown that the system performance can be improved by carefully adapting the bus cycle to the particular requirements of the application.

A broad historical perspective about real-time scheduling is presented in [SAA$^+$04].

Before discussing the types of scheduling, the entities that are considered in this thesis have to be characterized. These entities are tasks and messages.

2.1 Task model

A job is usually defined as a sequence of instructions to be executed by a processor. It is also known as a thread of execution or a (processor) scheduling unit. In real-time systems, a job has, at least, two parameters that are the release instant r_i and the worst-case computation time C_i. A task is a potentially infinite sequence of jobs where each job is a task instance. A task is periodic if it is time-triggered, with a regular release. The length of time between releases of successive instances of task τ_i is a constant, T_i, which is called the period of the task. Therefore, a periodic task τ_i can be completely characterized by the following three parameters: its worst-case computation time C_i, its period T_i and its relative

phase Ph_i, which determines the first release instant. Thus, the set of all synchronous tasks in the system can be expressed as:

$$\Gamma = \left\{ \tau_i \left(C_i, T_i, Ph_i \right), \forall i = 1, ..., m \right\}$$

(2-1)

Real-time jobs also have a temporal limit for finishing the execution that is called deadline D_i and that is relative to the job's release time. All jobs from a task have a common deadline. Most of the tasks executed in a distributed system either need data from other tasks or generate results to be used somewhere in the system, or both. A task that generates some data is called a producer task and a task that uses that data for any purpose is called a consumer task. For example, in a control loop built upon a distributed system, a task that acquires data from the outside world is a producer of sensor data, a controller task can consume this data and produce an actuation value and an actuator task, again, consumes this last data. When these tasks are in different nodes, as it is often usual, the produced data is conveyed in messages transmitted on a shared bus or network. The rest of the tasks do not interact with other tasks and thus they are called stand-alone tasks. In summary, the interaction between tasks can be classified according to 2 basic types:

- Stand-alone,
- Interactive.

In the first type are included the tasks that perform some kind of closed-loop control and don't communicate with other tasks. While the tasks that exchange data with other tasks are included in the second type.

Each interactive task communicates with other tasks in the system, using the message-passing paradigm, and can be decomposed to a simpler form where, at most, they produce and/or consume a single message:

- Producer, the task produces one message;
- Consumer, the task consumes one message;
- Producer/Consumer, the task consumes one message and produces another message.

It is assumed hereafter that the messages are sent at the end of the tasks execution and received at the beginning of each invocation.

In order to achieve a global synchronization, other parameters have to be defined, namely:

- D_i – The deadline measured relatively to the release instant;

14

- N_i – The node where the task runs;

And for interactive tasks also:

- MP_i – The message produced;

- MC_i – The message consumed.

Considering the new parameters, the set of all stand-alone synchronous tasks in the system can now be expressed as:

$$\Gamma = \{\tau_i(C_i, T_i, Ph_i, D_i, N_i), \forall i = 1,...,m\}$$

(2-2)

And the set of all interactive tasks can be denoted by:

$$\Gamma = \{\tau_i(C_i, T_i, Ph_i, D_i, N_i, MP_i, MC_i), \forall i = 1,...,m\}$$

(2-3)

An instance of a synchronous task is a particular execution of such a task. Apart from the parameters defined in the previous equations, an instance k of a task has two specific parameters that are the release instant $r_{i,k}$ and the absolute deadline $d_{i,k}$. The relation between two successive instances of a task τ_i is denoted by:

$$r_{i,k+1} = r_{i,k} + T_i \qquad \forall k = 1,...,nInst_i$$

(2-4)

where $nInst_i$ is the number of instances of task τ_i.

From equation (2-2), an extended set of parameters can be derived to define the set of instances of a stand-alone task. Thus, the new set becomes:

$$\Gamma = \{\tau_{i,k}(C_i, T_i, Ph_i, D_i, N_i, d_{i,k}, r_{i,k}), \forall k = 1,...,nInst_i, \forall i = 1,...,m\}$$

(2-5)

Whereas for interactive tasks, from equation (2-3), the new set becomes:

$$\Gamma = \{\tau_{i,k}(C_i, T_i, Ph_i, D_i, N_i, MP_i, MC_i, d_{i,k}, r_{i,k}), \forall k = 1,...,nInst_i, \forall i = 1,...,m\}$$

(2-6)

where $\tau_{i,k}$ is the instance k of the synchronous task τ_i. From this expression, the execution window can be defined as the interval between the release instant and the absolute deadline of a task.

15

Also a priority Pr_i could be defined for each task, in the case of fixed-priority scheduling, or for each job, in the case of dynamic priority scheduling. This is usually more useful for non real-time tasks where the deadlines don't play such an important role.

2.2 Message model

According to the communication paradigm selected for this study, every interactive task uses messages to exchange data with other tasks. Particularly, interactive tasks communicate through periodic messages. A periodic message is a potentially infinite sequence of message instances. In real-time systems, a message has, at least, two parameters that are the release instant r_j and the worst-case transmission time C_j. A periodic message σ_j can be completely characterized by the following three parameters: its worst-case transmission time C_j, its period T_j and its relative phase Ph_j, which determines the first release instant. Thus, the set of all synchronous messages in the system can be expressed as:

$$\Psi = \left\{ \sigma_j \left(C_j, T_j, Ph_j \right), \forall j = 1,...,n \right\}$$

(2-7)

In order to achieve a global synchronization, other parameters have to be defined, namely:

- D_j – The deadline measured relatively to the release instant;
- PT_j – Producer task;
- $CTL_{j,i}$ – Consumer task list.

Considering the new parameters, the set of all the synchronous messages in the system can be expressed as:

$$\Psi = \left\{ \sigma_j \left(C_j, T_j, Ph_j, D_j, PT_j, CTL_{j,i} \right), \forall j = 1,...,n, \forall i = 1,...,m \right\}$$

(2-8)

An instance of a synchronous message is a particular transmission of such a message. Apart from the parameters defined in equation (2-8), an instance k of a message has two specific parameters that are the release instant $r_{j,k}$ and the absolute deadline $d_{j,k}$. The relation between two successive instances of a message σ_j is denoted by:

$$r_{j,k+1} = r_{j,k} + T_j \qquad \forall k = 1,...,nInst_j$$

(2-9)

where $nInst_j$ is the number of instances of a message σ_j.

From equation (2-8), an extended set of parameters can be derived to define the set of instances of a message. The new set becomes as follows:

$$\Psi = \left\{ \sigma_{j,k}\left(C_j, T_j, Ph_j, D_j, PT_j, CTL_{j,i}, d_{j,k}, r_{j,k}\right), \forall k = 1,...,nInst_j, \forall j = 1,...,n, \forall i = 1,...,m \right\}$$

(2-10)

where $\sigma_{j,k}$ is the instance k of the synchronous message σ_j. From this expression, the transmission window can be defined as the interval between the release instant and the absolute deadline of a message.

Also a priority Pr_m could be defined for each message. This is usually more useful for non real-time messages where the deadlines don't play such an important role.

2.3 Scheduling real-time systems entities

A common classification [But00] of real-time task scheduling is done according to different aspects. These aspects will now be described:

- Off-line. All scheduling decisions are made prior to system execution.
- On-line. Scheduler decisions are taken during system runtime, upon the occurrence of some event that requires rescheduling.
- Static. Scheduling decisions are based on fixed information that is available at pre-runtime. In static task scheduling, all instances of a task have the same priority. Usually, the entities are numbered so that entity i has priority i, where the value one denotes the highest priority and larger integers denote lower priorities.
- Dynamic. Scheduling decisions are based on information that is available at runtime, only. In dynamic scheduling, the priority of each instance of an entity can be different.
- Non-preemptive. A running task executes until it decides to release the allocated resources, usually on completion, irrespectively of other tasks becoming read, eventually with higher priority. In this case, scheduling decisions are only required after task's completion instants.
- Preemptive. A running task can be suspended or interrupted during its execution, if at some instant a task with higher priority becomes ready.

The previous classification was used in the context of tasks, but it can also be used for real-time message scheduling.

2.4 Schedulability

A schedule is said to be feasible if all entities in a set meet their deadlines. In order to know this some type of schedulability testing has to be done. There are three classes of schedulability tests:

- Sufficient – passing it indicates that the entity set is schedulable
- Necessary – failing it indicates that the entity set is not schedulable
- Exact – passing indicates schedulability; failing indicates non-schedulability

It can be said that a scheduler is optimal, when it always finds a feasible schedule if one exists. Sufficient or necessary schedulability tests have an error margin but are simpler than exact ones, and sometimes the only reasonable solution. The simplest tests are utilization-based schedulability tests, which fail if the schedule will be using the CPU more than a certain percentage. An alternative approach is the response-time-based test, where the response time, or worst-case termination, of a task is calculated by adding to the WCET the interference of higher priority tasks. The process is repeated for each task. The schedulability test finalizes by simply comparing the computed with the desired maximum termination times. The response-time-based analysis has the advantage of being an exact test, and of giving a quantitative output. The same rationale can be used for messages considering their worst-case transmission time.

2.5 Examples of scheduling algorithms

This section briefly presents some paradigmatic scheduling algorithms.

The seminal work by Liu and Layland [LL73] includes two of the most important scheduling algorithms for independent task scheduling in single CPU systems. These algorithms are the Rate Monotonic, for systems with static priorities, and the Earliest Deadline First, for systems with dynamic priorities. The relevance of these algorithms results from the fact that they are optimal among their class. An algorithm is optimal if it is able to generate a feasible schedule whenever some other algorithm of the same class is able to do it.

2.5.1 Rate Monotonic algorithm

The Rate Monotonic (RM) algorithm [LL73] is an on-line preemptive algorithm based on static priorities. This algorithm is a simple rule that assigns priorities to entities according to their rates. Specifically, entities with higher request rates (that is, with shorter periods) will have higher priorities.

$$\forall \tau_i, \tau_j \in \Gamma : T_i < T_j \Rightarrow Pr_i > Pr_j$$

(2-11)

Since periods are constant, RM is a fixed-priority assignment: priorities are assigned to entities before execution and do not change over time. Moreover, RM is intrinsically preemptive: the currently executing entity is preempted by a newly arrived entity with shorter period.

In 1973, Liu and Layland [LL73] showed that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM.

2.5.2 Earliest Deadline First algorithm

The Earliest Deadline First (EDF) algorithm [LL73] is an on-line preemptive algorithm based on dynamic priorities. This algorithm selects the task with the earliest absolute deadline.

$$\forall \tau_{i,k}, \tau_{j,k} \in \Gamma_R : d_{i,k} < d_{j,k} \Rightarrow Pr_{i,k} > Pr_{j,k}$$

(2-12)

where Γ_R is the subset of Γ comprising the ready tasks and k is the number of the current instance of each task.

Tasks may arrive at any time and may be preempted.

2.5.3 Other scheduling algorithms

Many other scheduling algorithms have been developed along the years. Two other well-known algorithms are the Deadline Monotonic (DM) [LW82] and the Least-Laxity (LL) [MD78] algorithms.

3 Time-triggered communications

3.1 Physical processes

For a process to be controlled, a variable, or a set of variables, have to be monitored using some type of sensor. When a variable reaches some threshold value an alarm is activated. In order to bring the variable to a desired range, the alarm starts a control procedure that interacts with the process through an actuator. The process monitoring and control is accomplished by a process controller as in Figure 3-1.
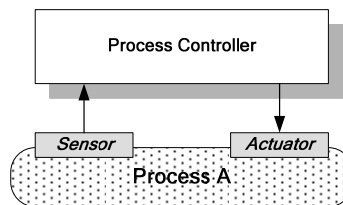
Figure 3-1 – Basic process monitoring and control

Each process is an external variable that should have, at least, three parameters:

- Desired interval,
- Tolerance intervals, and
- Maximum change rate, CR, (per time unit).

The desired interval is the range of values that do not need any system intervention. The tolerance intervals are the intervals of values where system feedback is required. In each tolerance interval, the value most distant from the desired interval is the critical value. Depending on the nature of the variable this critical value can be a soft, or a hard, deadline. Each of these intervals can be further separated into two intervals, namely:

- High tolerance interval, and
- Low tolerance interval.

The high tolerance interval defines a range of values that have as a limit a soft deadline and the low tolerance interval defines a range of values that have as a limit a hard deadline.
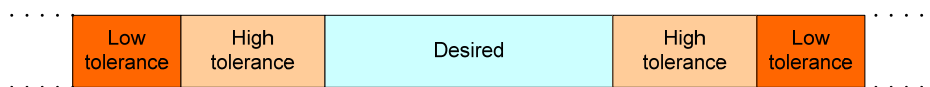
Figure 3-2 – Tolerance intervals

A node (depicted by a rectangle) represents a processing unit with private memory. All the nodes are connected with a common bus. At each node there is, at least, a task that produces and/or consumes one or more messages. In this simple example there are several messages that are transmitted across the bus, like the message from *Data Acquisition A* to *Controller A*.

From the system side we can have three parameters:

- Response time (from the instant where the variable leaves the desired interval until the system reacts),
- Execution time, and
- Change rate, CR, (per time unit).

It should be noted that the system CR must be greater than the controlled variable maximum CR, otherwise there is no guarantee that the variable is outside the desired interval for a bounded period of time.

The process parameters and the system parameters lead us to the definition of the system's tasks and messages deadlines.

A certain variable is controlled by a set of tasks, related by a precedence graph, that are allocated in various nodes and that the last task in the precedence graph is responsible for manipulating the actuator. In order to define the parameters of the set of tasks, a relation between them and the process variable's parameters must be found.

If there are several processes that need a similar control then a process controller is coupled to each process. This process control can be successfully accomplished through a distributed embedded system.

A typical distributed control system acts upon a process by acquiring a signal, processing some data and responding through an actuator. Figure 3-3 represents 2 processes, A and B, being controlled by a distributed control system.
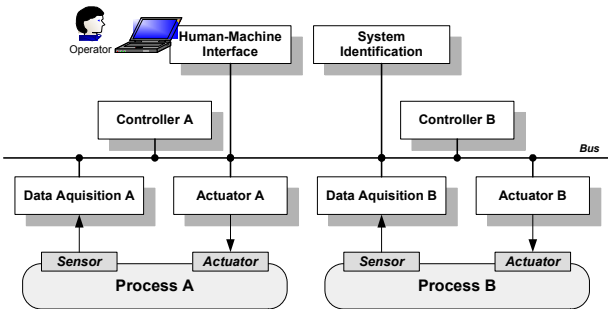
Figure 3-3 – Typical distributed control system.

Distributed embedded systems used in real-time applications such as distributed control systems or automotive systems are often based on a set of nodes connected by a common bus. In many of these systems, the nodes are based in small 8/16 bit microprocessors or microcontrollers, thus having reduced processing power.

The advantages of using a distributed embedded system and not a centralized approach have to do mainly with a better resource usage. Due to an open, and well known, interface between the nodes, system operation can be expanded through the integration of new functionalities or fault tolerance.

Distributed Embedded Systems (DES) are a typically part of intelligent automatic equipment with a high degree of autonomy. In most cases, DES have a strong impact on human lives, either because they are used within important economic processes, e.g. complex machinery in factories, or because they control equipment that directly interacts with people, e.g. transportation systems [BS05].

The importance of DES has been growing steadily and it is expected to grow even further as distribution provides an efficient way to improve several desirable properties in a system, from maintainability, to scalability, composability and dependability, to name a few [Kop97] [Koo02]. Also, DES are a natural support for higher integration of resources in complex systems, e.g. robots, cars and planes, with a potential for lower costs and lower overall complexity [Rus01].

However, the positive aspects of distribution do not come for granted. Thus specific techniques and protocols must be used to achieve the desired properties. Therefore, designing and deploying such techniques and protocols is still an important research topic [BS05].

3.2 Real-time communication

There are two basic communication paradigms: event-triggered and time-triggered. The event-triggered paradigm refers to the communication where an application sends messages upon the occurrence of some event, such as a change in the value of some input. On the other hand, according to the time-triggered paradigm, messages are sent only in precise pre-defined time instants.

Presently, most distributed systems, either industrial or embedded, rely in a serial communications infrastructure known as fieldbus [Tho98] to interconnect a set of nodes. Some of the available fieldbuses are presented in the following sections.

3.3 The Controller Area Network protocol

The Controller Area Network [Rob91] (CAN) protocol was developed in the mid 1980s by Robert Bosch GmbH, aiming at automotive applications, to provide a cost-effective communication bus for in-car electronics and as an alternative to expensive and cumbersome wiring looms. It is standardized as ISO 11898-2 [Int93] for high speed applications (1Mbps) and ISO 11519-2 [Int94] for lower speed applications (125Kbps). The network maximum length depends on the data rate. Due to its bitwise arbitration mechanism, it is required that the bit time must be long enough to allow the signal propagation along the entire network as well its decoding by other stations, which imposes a fundamental limit to the maximum speed attainable (e.g. 40m @ 1Mbps).

The CAN protocol has become the main fieldbus used in the time critical parts of automotive systems. CAN operates following the event-triggered paradigm.

CAN has the following properties:

- Prioritization of messages,

- Guarantee of latency times,

- Configuration flexibility,

- Multicast reception with time synchronization,

- System wide data consistency,

- Multimaster,

- Error detection and signalling,

- Automatic retransmission of corrupted messages as soon as the bus is idle again,

- Distinction between temporary errors and permanent failures of nodes and autonomous switching off of defect nodes.

In real-time message scheduling, the computation of the transmission time of messages is of paramount importance, since it is required to perform any kind of analysis. To provide clock information embedded in the bit stream, CAN uses bit-stuffing. This implies that the actual number of transmitted bits not only can be larger than the size of the original frame, but also can vary in consecutive instances of the same message, depending on the

particular message instance contents. According to the CAN standard [Rob91], the total number of bits in a CAN frame without bit-stuffing is given by equation (3-1), where DLC is the number of bytes of payload data in a CAN frame ([0,8]) and 47 is the number of control bits.

$$CAN\_LEN_{WithoutStuffing} = 47 + 8 \times DLC$$

<div align="right">(3-1)</div>

The CAN frame layout is defined such that only 34 of these 47 bits are subject to bit-stuffing. Therefore the worst-case number of bits after bit-stuffing is given by equation (3-2) [NHNP01].

$$CAN\_LEN_{WithStuffing} = 47 + 8 \times DLC + \left\lfloor \frac{34 + 8 \times DLC - 1}{4} \right\rfloor$$

<div align="right">(3-2)</div>

3.4 TTCAN

The possibility of using a time-triggered approach in part of the operation of CAN based distributed systems was explored by the standard ISO11898-4 [Int00], and is known as Time-Triggered Controller Area Network (TTCAN). The TTCAN [FMD$^+$00] defines a new session layer for CAN.

The TTCAN protocol is a development in CAN technology that specifies a time-slot based communication mechanism. TTCAN technology avoids the transmission collisions commonly found in standard CAN networks. In TTCAN, all the message instances are transmitted only on previously allocated time-slots. No other instance may be transmitted on an allocated time-slot, so transmission collisions are avoided.

The TTCAN goals are to reduce latency jitters, guarantee a deterministic communication pattern on the bus and use the physical bandwidth of a CAN network more efficiently.

In TTCAN, a special node, the time master, is responsible for the transmission of a systolic message, the reference message, which is used to achieve synchronization between the fieldbus nodes. This allows the introduction of a system wide global network time with high precision. Based on this time the different messages are assigned to time windows within a basic cycle. The reference message marks the start of a time slot called the Basic Cycle (BC). The BC may be divided in different types of windows, namely, the exclusive

windows and the arbitrating windows. An exclusive window is used to transmit a specific periodic message. An arbitrating window may be used to transmit any message, provided it gains the bitwise arbitrating process as in normal CAN operation and provided it finishes transmission before the end of the window, thus guaranteeing temporal isolation between the different types of traffic.

The complete traffic pattern in a TTCAN system consists in a fixed number of consecutive BCs and is named the System Matrix or Matrix Cycle (SM). A major restriction in the SM construction is the column like organization. All the windows of the same column in every BC must be of the same size. With this organization it is possible to define a set of trigger instants (offsets from the reference message) which is kept constant from BC to BC all along the SM, thus simplifying the TTCAN controller hardware. Once the SM is defined, it is possible to merge two (or more) consecutive arbitrating windows in the same BC in order to facilitate the transmission of normal CAN messages. Finally, the other major restriction is in the number of BCs per SM, which must be an integer power of two.

In TTCAN, the system matrix must be defined off-line. All the nodes must have stored the correspondent information prior to the start of operation.

A big advantage of TTCAN compared to classic scheduled systems is the possibility to transmit event-triggered messages in certain "arbitrating" time windows as well. These time windows, where normal arbitration takes place, allow the transmission of spontaneous messages.

3.5 TTP/C

The TTP/C [Kop99a] protocol is a reliable and fault-tolerant communication protocol, designed to permit high performance data transmission, clock synchronization, membership services, fast error detection and consistency checks. A TTP/C network consists of a set of communicating nodes connected by a replicated interconnection network. A node computer comprises a host computer and a TTP/C communication controller with two bi-directional communication ports. Each of these ports is connected to an independent channel of a dual-channel interconnection network. Via these broadcast channels the nodes communicate using the services of the communication controller.

The TTP/C protocol implements broadcast communication that proceeds according to an *a priori* established time-division multiple access (TDMA) scheme. This TDMA scheme

divides time into slots, each being statically assigned to a particular node, and, during its slot, each node has exclusive write permission to the network. The slots are grouped in the so-called TDMA rounds. In a TDMA round every node is granted write permission in at least one slot, and the access pattern repeats itself in successive rounds.

A distributed fault-tolerant clock synchronization algorithm establishes the global time base needed for the distributed execution of the TDMA scheme. Nodes can send different messages in different TDMA rounds, although the slot length is constrained to be the same. To handle this feature, the protocol defines cluster cycles, comprising several TDMA rounds with all the possible message combinations.

Message scheduling in TTP/C is performed at pre-runtime, which turns out this protocol unsuited to handle dynamic message sets. Nevertheless, a limited degree of flexibility still exists, both due to the possibility of pre-configuring several modes of operation and to the possibility of reserving TDMA slots for later expansion.

3.6 FTT-CAN

The basis for the FTT-CAN protocol (Flexible Time-Triggered communication on CAN) has been first presented in [AFF98]. Basically, the protocol makes use of the dual-phase elementary cycle concept in order to combine time and event-triggered communication with temporal isolation. The time-triggered traffic is scheduled on-line and centrally in a particular node called master. This feature facilitates the on-line admission control of dynamic requests for periodic communication because the respective requirements are held centrally in just one local table. With on-line admission control, the protocol supports the time-triggered traffic in a flexible way, under guaranteed timeliness (dynamic planning-based scheduling paradigm). The FTT-CAN protocol follows a network-centric approach [AF00] because it considers the fieldbus as the pivot element in the interactions among nodes. Due to this it facilitates the on-line control of distributed applications.

The FTT-CAN takes advantage of the native MAC of CAN to reduce communication overhead and to support a high efficiency and flexibility in the time-triggered traffic. The protocol relies on a relaxed master-slave medium access control in which the same master message triggers the transmission of messages in several slaves simultaneously (master/multi-slave). The eventual collisions between slaves' messages are handled by the native distributed arbitration of CAN. The protocol also takes advantage of the CAN

arbitration to handle event-triggered traffic in the same way as the original protocol does. Particularly, there is no need for the master to poll the slaves for pending event-triggered requests. Slaves with pending requests may try to transmit immediately, as in normal CAN, but just within the respective phase of each elementary cycle. This scheme allows a very efficient combination of time and event-triggered traffic, particularly resulting in low communication overhead and shorter response times.

The nomenclature used in the protocol follows. In FTT-CAN the bus time is slotted in consecutive Elementary Cycles (ECs) with fixed duration. All nodes are synchronized at the start of each EC by the reception of a particular message known as *EC trigger message* (TM), which is sent by a particular node called *master*.

Within each EC the protocol defines two consecutive windows, asynchronous and synchronous, that correspond to two separate phases (see Figure 3-4).
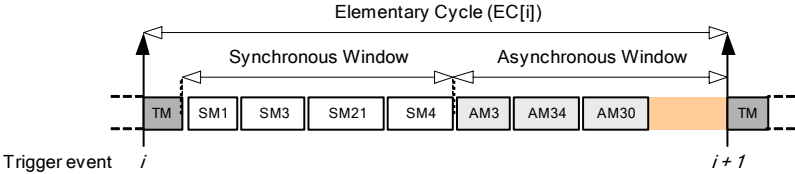
Figure 3-4 – The Elementary Cycle in FTT-CAN

The former one is used to convey event-triggered traffic, herein called *asynchronous* because the respective transmission requests can be issued at any instant. The latter one is used to convey time-triggered traffic, herein called *synchronous* because its transmission occurs synchronously with the ECs. The schedule for each EC is conveyed by the respective EC trigger message. Since this window is placed at the end of the EC, its starting instant is variable and it is also encoded in the respective EC trigger message. The asynchronous window has a duration that is equal to the remaining time between the EC trigger message and the synchronous window.

The communication requirements are held in a database located in the master node [Ped03], the System Requirements Database (SRDB). The SRDB holds the properties of each of the message streams to be conveyed by the system, both real-time and non-real-time, as well as a set of operational parameters related to system configuration and status. This information is stored in a set of three tables: the Synchronous Requirements Table, the Asynchronous Requirements and the Configuration and Status Record (SCSR). Based on the SRT, an on-line scheduler builds the synchronous schedules for each EC.

From an operational point of view, two different solutions have been used to implement the scheduler. One is the planning-scheduler [APF99] and the other makes use of FPGA-based scheduling co-processors [MF01].

An in-depth analysis of the FTT-CAN protocol is available in [APF02].

FTT-CAN impairments to dependability, namely the single point of failure formed by the master node and the fail uncontrolled nature of current FTT-CAN nodes, were identified and discussed in [Fer05]. He has presented a fault-tolerant system architecture with replicated masters and fail-silent nodes. The specific problems and mechanisms related with master replication, particularly a protocol to enforce consistency during updates of replicated data structures and another protocol to transfer these data structures to an unsynchronized node upon asynchronous start-up or restart, were also addressed.

4 Information flow and holistic scheduling

The application domain of this study is real-time distributed systems, where various computational nodes are interconnected by a network. In each node several communicating tasks can be allocated. This chapter focuses into two aspects of this type of systems that are the information flows between tasks and the holistic scheduling of tasks and messages. The first aspect studies types of task interaction. From here a methodology for the representation of these interactions is presented. The second aspect makes use of the analysis of the task interaction to support the parameter determination and tuning of tasks and messages towards a holistic scheduling. The parameter determination is based on new scheduling options like the approach, either network or node centric, and the relation between the execution and transmission windows of related tasks and messages. Another topic that is discussed is the concept of real-time procedures. A procedure is implemented by a number of real-time tasks that communicate using messages. A procedure can define some parameters that specify the restrictions to be globally used with the corresponding set of tasks and messages. The transposition of the procedure's parameters is guided by the same aspects as the approach and the relation of windows.

4.1 Information flow

The design of real-time software must incorporate all of the fundamental concepts associated with high-quality software like abstraction and modularity. In addition, real-time software poses a set of unique problems for the designer, such as:

- Representation of interrupts and context switching,
- Concurrency as manifested by multitasking and multiprocessing,
- Intertask communication and synchronization,
- Wide variations in data and communication rates,
- Representation of timing constraints,
- Asynchronous processing,
- Necessary and unavoidable coupling with operating systems, hardware, and other external system elements.

Several real-time software design methodologies have been proposed to grapple with some or all of the problems noted above. Some design methods extend one of the three classes of

design, namely: data flow [WM86], data structure [Jac83], or object-oriented [Boo93] methodologies. Others introduce an entirely separate approach, using finite state machine models or message passing systems [Wit85], Petri nets [Vid83], or a specialized language [Ste84] as a basis. Some of these techniques will now be briefly presented (based on [Lap97]).

Flowcharts

Flowcharts are probably the oldest modelling tool for software systems, and are widely understood. For this reason, simple systems can be modelled with flowcharts. In multitasking systems, flowcharts can be used to describe each task separately, but the interaction between processes is not easily represented, and temporal behaviour cannot be described.

Structure charts

Structure charts or calling tree are a widely used mechanism for describing the modular decomposition of a system. In structure charts, rectangles are used to represent processes. Moving from left to right in the structure chart, it represents increasing sequence in execution. Moving from top to bottom along any branch of the structure chart indicates increasing detail. There is no formal method for indicating conditional branching in the tree. Structure charts have been used as long as flowcharts to describe system functionality, and like flowcharts, they are useful for very simple systems. Structure charts have several advantages: they clearly identify function execution sequence; they help to identify recursion and repeated modules; and they encourage top-down design. However, structure charts are not useful in the description of concurrent systems.

Finite state machines

Finite state machines (FSM), or finite state automata, are a type of mathematical model used in the design of compilers, hard-wired logic, and communication systems, among others. FSMs rely on the fact that many systems can be represented by a fixed number of unique states. The system may change state depending on time or the occurrence of specific events. There are two kinds of FSM. Firstly a Moore FSM consists of a nonempty, finite set of states, one of which is the initial state, and one or more are terminal states. An alphabet of symbols is given, and a transition function shows how, given a current state

and symbol, a new state is entered. The Moore machine does not allow for outputs during transition, but output is provided for by the processes represented by states. Outputs during transition are allowed, however, by a variation of the Moore machine called a Mealy FSM. In summary, FSMs are widely used in the specification of systems that are state driven. Concurrency can be depicted by using multiple FSMs. On the other hand, the major disadvantage of FSMs is that there is no way to indicate how functions can be broken down into sub-functions. In addiction, intertask communication for multiple FSMs is difficult to depict.

Dataflow diagrams

Data-flow models are used to show how data flows through a sequence of processing steps. The data is transformed at each step before moving on to the next stage. These processing steps, or transformations, are program functions when data-flow diagrams are used to document a software design. Extensions to data flow representations that provide the mechanics for real-time software design have been proposed. Hassan Gomaa [Gom84] proposed one extension called Design Method for Real-Time Systems (DARTS). This extension allows real-time system designers to adapt data flow techniques to the special needs of real-time applications. In the end, dataflow diagrams are well understood and widely used. Dataflow diagrams are usually combined with some of the other techniques to yield a coherent software requirements document. The only major weakness that seems to be inherent in dataflow diagrams is that they make it difficult to depict synchronization in flow.

Petri nets

The concept of Petri net has its origin in [Pet62]. Petri nets are another type of mathematical model used to specify the operations to be performed in a multiprocessing or multitasking environment, but they can also be described graphically. Graphical Petri nets have representation for data stores, processes, transitions and operations. The processes and transitions are labelled with a data count and transitions function, respectively, and are connected. In Petri nets, the graph topology does not change over time; only the "markings" or connects of the places (represent data stores or processes) do. The system advances as transitions are "fired". Petri nets can be used to model systems and to analyze timing constraints and race conditions. Petri nets are excellent for representing

multiprocessing and multiprogramming systems, especially where the functions are simple. Because they are mathematical in nature, techniques for optimization and formal program-proving can be employed. According to Laplante [Lap97], Petri nets can be overkill if the system is too simple and timing can become obscured when the system is highly complex. Several time-related extensions of Petri nets were proposed. They primarily impose additional timing constraints onto transitions or places. The imposed timing constraints can be represented as constants or functions.

Statecharts

Harel's statecharts [Har87] [Har88] combine FSM with dataflow diagrams and a feature called broadcast communications in a way that can depict synchronous and asynchronous operations. Statecharts allow a top-down design through levels of detail. Each of the levels can in turn be decomposed into appropriate states that represent program modules, or procedures. They allow the representation of concurrency where synchronization can be achieved through broadcast communication. This allows various processes to change state due to a common event. In short, Hare's statecharts are good for representing real-time systems because they can easily depict concurrency while preserving modularity. In addition, the concept of broadcast communication allows for easy intertask communication representation.

Other approaches to the design of real-time systems include the Unified Modelling Language and the Architecture Description Languages. These will now be briefly presented.

Unified Modelling Language

The Unified Modelling Language (UML) is an object modelling and specification language used in software engineering [RJB98]. The application of UML to the development of real-time systems is covered in [Dou04]. The object-based UML can describe the structural and behavioural aspects critical to real-time systems and has come to the fore as an outstanding medium for effective design. But UML is a rather heavyweight design notation, modelling the full structure and semantics of a software system in seven separate views.

Architecture Description Languages

Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components and connectors) and their overall interconnection structure [MT00]. Generally, software architectures are composed of components, connectors and configurations. Components are the set of computation units. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Finally, configurations are connected graphs of components and connectors that describe architectural structure. To support architecture-based development, formal modelling notations and analysis and development tools that operate on architectural specifications are needed. Architecture Description Languages (ADLs) and their accompanying toolsets have been proposed as the answer. Loosely defined, "an ADL for software applications focuses on the high-level structure of the overall application rather than on the implementation details of any specific source module" [Ves93].

A number of ADLs have been proposed for modelling software architectures both within a particular domain and as general-purpose architecture modelling languages. These ADLs can be divided into first-generation ADLs and XML-based ADLs [DHT01]. First-generation ADLs are characterized by proprietary language syntaxes, while XML-based ADLs benefit from the extensible nature of XML standard [YBP+04] [YCB+04]. Representatives of the first-generation software ADLs are: ACME, Rapide, Unicon and Wright. The first-generation software ADLs are thoroughly classified and compared in [MT00]. Some examples of XML-based ADLs are: ADML and xADL. Recent XML-based ADLs are presented in [Spe00] and [DHT01]. The modification, or extension, of software ADLs can be hard due to one of two reasons, either it is custom-tailored to support only one great feature, but not covering the needs of this study, or it is so general that a lot of changes are required in order to have only the required features.

Another approach to the design of real-time systems is the use of real-time procedures that include timing constraints. These procedures may be organized with a data flow-oriented method based on simple data streams. A data stream shows the information flow between tasks. The tasks follow the producer-consumer model. Data flows from one producer task to one or more consumer tasks. Consumer tasks may also produce data to other tasks thus

becoming consumer/producer tasks. This way, a data stream represents only the tasks and data that make use of critical resources like the processing units and the network. This method differs from other approaches, mainly in the way the transitions between tasks are specified. These transitions don't specify the triggering conditions but follow a simpler approach more suited to distributed control systems. In this approach only the possible sets of messages to be transmitted has to be specified. This way, the internal computational aspects of tasks remain encapsulated. Also, in this approach both tasks and messages are the main entities, therefore bringing messages to the same level of importance.

The referred approach is only focused in aspects relevant to the parameter determination of entities towards an automated scheduling. When comparing to the UML, the proposed approach is geared toward lightweight experimentation and easy extension. In ADLs, a component refers to a unit of computation, while in this study both the tasks and the messages are the main entities, therefore bringing messages to the same level of importance. The proposed approach is only focused in aspects relevant to the parameter determination of entities towards an automated scheduling.

The result of this approach is a set of data streams, where tasks and messages have some parameters derived. These parameters will make possible a holistic system scheduling of all entities. Also, the data stream analysis is shown to be beneficial in different phases of a distributed system planning.

As shown in the following sections, the data stream based approach also provides a good support to the analysis of the interactions between tasks. With this methodology we are looking for a global synchronization of tasks and messages so that each of them gets the resources when needed.

4.1.1 Interactions between tasks

The most basic form of intertask interaction in distributed systems is message exchange. This enables a sending task, the producer task, to transmit a single message to a receiving task, the consumer task. This is the communication paradigm used for intertask interaction. Message passing between a pair of tasks can be supported by two message communication operations: *send* and *receive,* defined in terms of destinations and messages. In order for one task to communicate with another, one task sends a message (a sequence of bytes) to a

destination and another task, at the destination, receives the message. This activity involves the communication of data from the sending task to the receiving task and may involve the synchronization of the two tasks. Therefore, the tasks follow the producer-consumer model.

Several data flow scenarios may occur in an ordinary system. The data flow scenarios can be divided, from the point of view of the communication, into:

- Unicast,
- Multicast.

They can also be divided, from the point of view of the optional transmission of messages, into:

- Required transmission,
- Optional transmission.

And finally, they can also be divided, from the point of view of the set of messages to transmit, into:

- Fixed set,
- Variable set.

The combinations that will be considered in this study are depicted in Table 4-1. These refer to both unicast and multicast communication. The total number of identified scenarios is twenty. The combination of an optional transmission with a fixed set of messages to transmit does not make sense.

	Required transmission	Optional transmission
Fixed set	• Single message • Multiple messages	N. A.
Variable set	• Single message • Multiple messages • Single set of messages • Multiple sets of messages	• Single or no message • Multiple or no messages • Single or no set of messages • Multiple or no sets of messages

Table 4-1 – Data flow scenarios that were identified

The optional transmission of messages is substantially different from the required transmission of messages because the first refers to non-periodic tasks or messages while

the second refers only to periodic tasks and messages. For the purposes of this study only the periodic tasks and messages are considered.

Data flows from one producer task to one or more consumer tasks. Consumer tasks may also produce data to other tasks thus becoming consumer/producer tasks. Naturally a producer task also has some type of input data, but this data is not represented because the focus is on data that might be transmitted across the network. The same happens with a consumer task that also has some type of output data. This way, the task interaction scenarios represents only the tasks and data that make use of critical resources like the processing units and the network.

Several data flow scenarios may occur in an ordinary system. The fixed set scenarios, both unicast and multicast, represent the basic scenarios while the variable set scenarios, both unicast and multicast, represent the expansion scenarios. There are four basic scenarios and sixteen expansion scenarios.

The first two basic scenarios refer to unicast of single, or multiple messages, and are depicted in Figure 4-1.
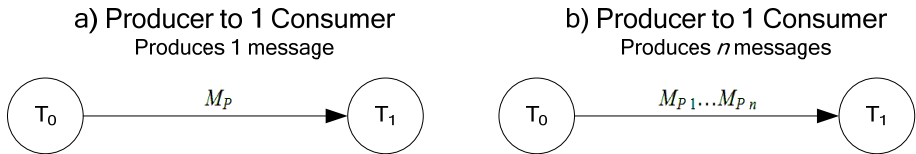
Figure 4-1 – Unicast of single or multiple messages

The basic interaction between tasks is shown in Figure 4-1.a). In this scenario, task T_0 produces message M_P to task T_1.

A producer task can unicast several messages, M_{P1} to M_{Pn}, to another task as shown in Figure 4-1.b). If all messages have the same properties, apart from the message size, than this scenario is just an extension of the previous one.

Figure 4-2 depicts the other two basic scenarios that refer to multicasting of single or multiple messages.
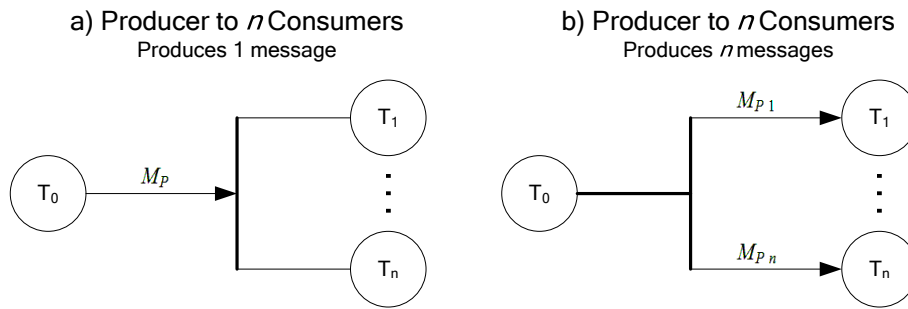
Figure 4-2 – Multicast of single or multiple messages

Multicasting of a single message to several tasks is shown in Figure 4-2.a). The combination of the multicast property with the possibility of sending several messages results in the situation shown in Figure 4-2.b). This scenario makes possible sending different messages to different tasks.

The first two expansion scenarios that show unicasting of a single message from a set of messages are depicted in Figure 4-3.
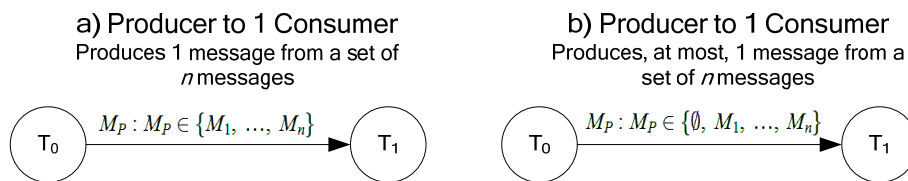
Figure 4-3 – Unicast of a single message from a set of messages to a task

Figure 4-3.a) shows task T_0 producing one message to Task T_1. This message is one of a set of possible messages to be transmitted. In this scenario it is reasonable to demand that all messages have the same properties apart from the message size. This way the transmission time is calculated using the message with the largest size.

The possibility of not transmitting any message is considered in Figure 4-3.b). In this situation, the producer task can produce one message from the set $\{M_1,\ldots,M_n\}$ or not produce any message, which is represented by the null element in the beginning of the set. Because this scenario has a non-periodic produced message it should be considered as asynchronous communication.

Two more expansion scenarios that refer to the unicast of a single set of messages from a set of sets of messages are depicted in Figure 4-4.

a) Producer to 1 Consumer
Produces 1 set of n messages from a set of i sets

$$S\,M_P : S\,M_P \in \{S\,M_1, \ldots, S\,M_i\}$$
$$S\,M_i = \{M_{i\,1}, \ldots, M_{i\,n}\}$$

b) Producer to 1 Consumer
Produces, at most, 1 set of n messages from a set of i sets

$$S\,M_P : S\,M_P \in \{\emptyset, S\,M_1, \ldots, S\,M_i\}$$
$$S\,M_i = \{M_{i\,1}, \ldots, M_{i\,n}\}$$
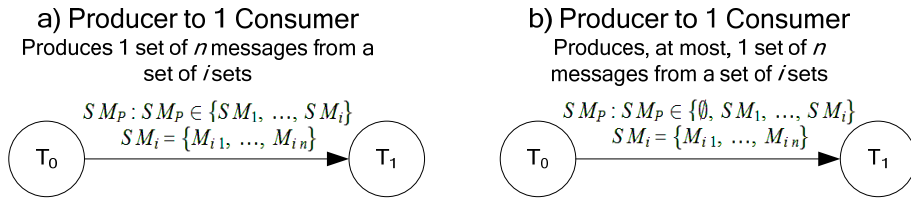
Figure 4-4 – Unicast of a single set of messages from a set of message sets

In these scenarios one set of messages is transmitted to a consumer task. This set of messages is one of the possible sets that are grouped as a set of sets. In Figure 4-4.b) the possibility of not sending any set of messages is also considered.

Two other expansion scenarios that refer to the unicast of multiple messages from a set of messages are depicted in Figure 4-5.

a) Producer to 1 Consumer
Produces m messages from a set of n messages

$$M_{P\,1} \ldots M_{P\,n} : M_{P\,n} \in \{M_1, \ldots, M_n\}$$

b) Producer to 1 Consumer
Produces m or no messages from a set of n messages

$$M_{P\,1} \ldots M_{P\,n} : M_{P\,n} \in \{\emptyset, M_1, \ldots, M_n\}$$
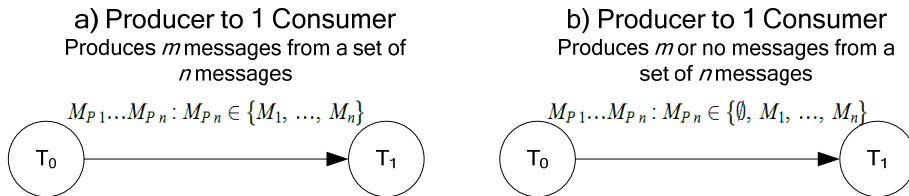
Figure 4-5 – Multicast of a single message from a set of messages

Figure 4-5.a) shows the situation of a unicast of multiple messages M_{P1} to M_{Pn} from a set of messages, $M_1 \ldots M_m$. A fixed number of messages from the set are transmitted after each execution of task T_0. Figure 4-5.b) considers the unicast of a variable number of messages from the set of messages $M_1 \ldots M_m$. It is possible not to transmit a message at all. Because this scenario has a non-periodic produced message it should be considered as asynchronous communication.

Two further scenarios that refer to unicasting of multiple sets of messages from a set of sets of messages are depicted in Figure 4-6.

a) Producer to 1 Consumer
Produces m sets of n messages from a set of i sets

$$S\,M_{P\,1} \ldots S\,M_{P\,n} : S\,M_{P\,n} \in \{S\,M_1, \ldots, S\,M_i\}$$
$$S\,M_i = \{M_{i\,1}, \ldots, M_{i\,n}\}$$

b) Producer to 1 Consumer
Produces m or no sets of n messages from a set of i sets

$$S\,M_{P\,1} \ldots S\,M_{P\,n} : S\,M_{P\,n} \in \{\emptyset, S\,M_1, \ldots, S\,M_i\}$$
$$S\,M_i = \{M_{i\,1}, \ldots, M_{i\,n}\}$$
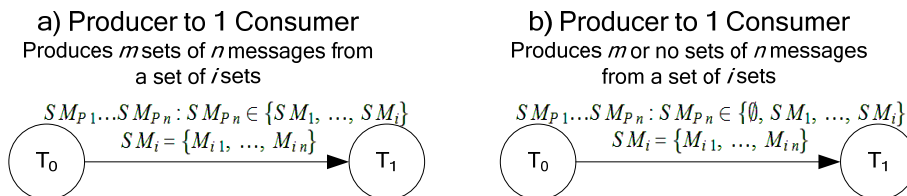
Figure 4-6 – Multicast of a single set of messages from a set of message sets

This scenario is basically the combination of the scenarios depicted in Figure 4-4 and Figure 4-5.

Two other expansion scenarios that refer to the multicast of a single message from a set of messages are depicted in Figure 4-7.

a) Producer to m Consumers
Produces 1 message from a set of n messages

$M_P : M_P \in \{M_1, \ldots, M_n\}$

T_0 → T_1 ⋮ T_m

b) Producer to m Consumers
Produces, at most, 1 message from a set of n messages

$M_P : M_P \in \{\emptyset, M_1, \ldots, M_n\}$

T_0 → T_1 ⋮ T_m
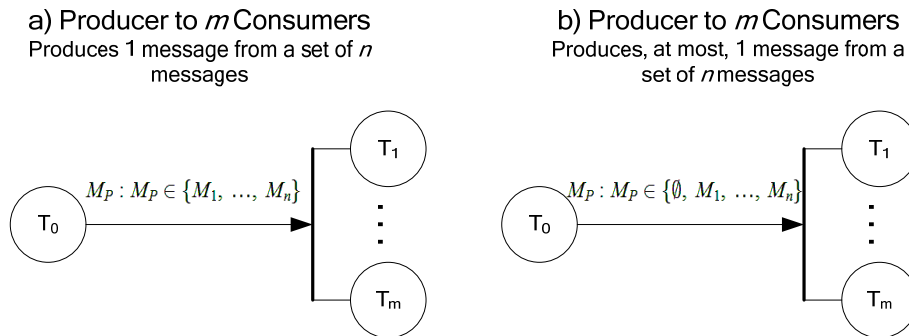
Figure 4-7 – Multicast of multiple messages from a set of messages

Two further scenarios that refer to multicasting of a single set of messages from a set of sets of messages are depicted in Figure 4-8.

a) Producer to m Consumers
Produces 1 set of n messages from a set of i sets

$SM_P : SM_P \in \{SM_1, \ldots, SM_i\}$
$SM_i = \{M_{i\,1}, \ldots, M_{i\,n}\}$

T_0 → T_1 ⋮ T_m

b) Producer to m Consumers
Produces, at most, 1 set of n messages from a set of i sets

$SM_P : SM_P \in \{\emptyset, SM_1, \ldots, SM_i\}$
$SM_i = \{M_{i\,1}, \ldots, M_{i\,n}\}$

T_0 → T_1 ⋮ T_m
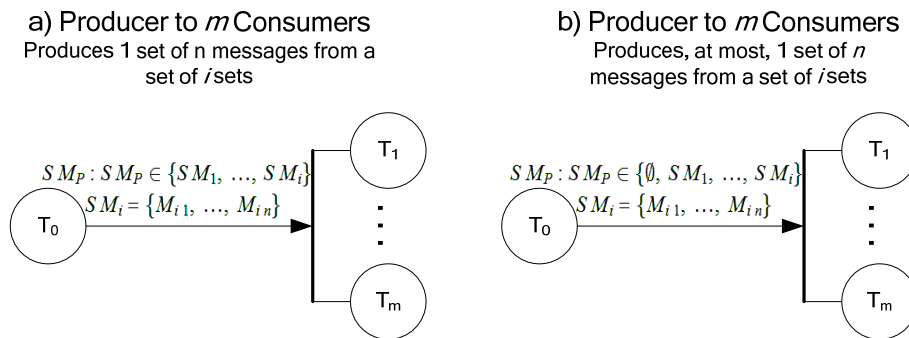
Figure 4-8 – Multicast of sets of messages from a set of message sets

Two other scenarios that refer to the multicasting of multiple messages from a set of messages are presented in Figure 4-9.

a) Producer to m Consumers
Produces m messages from a set of n messages

$M_{P\,1} : M_{P\,1} \in \{M_{11}, \ldots, M_{1\,n}\}$

T_0 → T_1 ⋮ T_m

$M_{P\,m} : M_{P\,m} \in \{M_{m\,1}, \ldots, M_{m\,n}\}$

b) Producer to m Consumers
Produces, at most, m messages from a set of n messages

$M_{P\,1} : M_{P\,1} \in \{\emptyset, M_{11}, \ldots, M_{1\,n}\}$

T_0 → T_1 ⋮ T_m

$M_{P\,m} : M_{P\,m} \in \{\emptyset, M_{m\,1}, \ldots, M_{m\,n}\}$
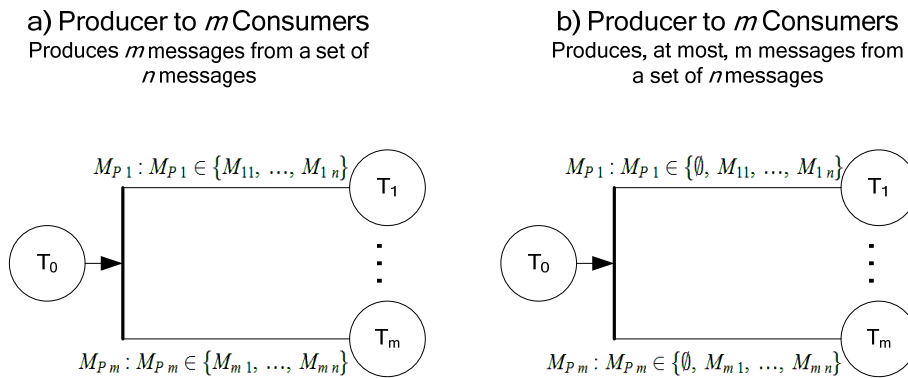
Figure 4-9 – Multicast of multiple messages from sets of messages

Finally, the two remaining expansion scenarios that refer to the multicasting of sets of messages from a set of sets of messages are shown in Figure 4-10.

a) Producer to m Consumers
Produces m sets of n messages in a set of i sets

$$SM_{P1} : SM_{P1} \in \{SM_{11}, \ldots, SM_{1\,i}\}$$
$$SM_{1\,i} = \{M_{1\,i\,1}, \ldots, M_{1\,i\,n}\}$$

T_1

T_0

T_m

$$SM_{P\,m} : SM_{P\,m} \in \{SM_{m\,1}, \ldots, SM_{m\,i}\}$$
$$SM_{m\,i} = \{M_{m\,i\,1}, \ldots, M_{m\,i\,n}\}$$

b) Producer to m Consumers
Produces, at most, m sets of n messages in a set of i sets

$$SM_{P1} : SM_{P1} \in \{\emptyset, SM_{11}, \ldots, SM_{1\,i}\}$$
$$SM_{1\,i} = \{M_{1\,i\,1}, \ldots, M_{1\,i\,n}\}$$

T_1

T_0

T_m

$$SM_{P\,m} : SM_{P\,m} \in \{\emptyset, SM_{m\,1}, \ldots, SM_{m\,i}\}$$
$$SM_{m\,i} = \{M_{m\,i\,1}, \ldots, M_{m\,i\,n}\}$$
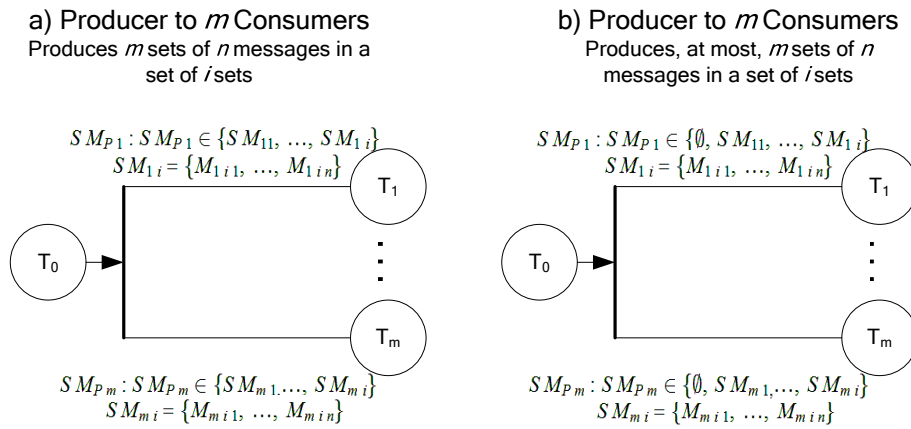
Figure 4-10 – Multicast of multiple sets of messages from sets of message sets

With these basic scenarios many others can be constructed. A complex program with communicating modules can be translated into a task composed of a set of communicating subtasks where each subtask has resource requirements, involves the execution of sequential code, and has communication as well as precedence constraints with other subtasks [Ram95].

A common restriction to all scenarios is that any message consumed has to be available in the beginning of the task execution and any produced message is only available after the end of the task execution.

The study and representation of these interactions is very important to the holistic scheduling.

4.1.2 Data streams

From the task interaction scenarios of a given system, potentially, several data streams may be identified [CF05]. A data stream shows an information flow between tasks. Data flows from one producer task to one or more consumer tasks. Consumer tasks may also produce data to other tasks thus becoming consumer/producer tasks. Therefore, a data stream begins with a producer task and ends with a consumer task. Naturally, a producer task also has some type of input data, but this data is not represented because the focus is on data that might be transmitted across the network. The same happens with a consumer task that

also has some type of output data. This way, a data stream represents only the tasks and data that make use of critical resources like the processing units and the network.

The representation of a data stream is very close from the one used for the task interactions. The main difference is that, in the case of the data streams, each task may only consume from another task and may only produce to another task. An example of data streams derived from a task interaction scenario is presented in Figure 4-11.
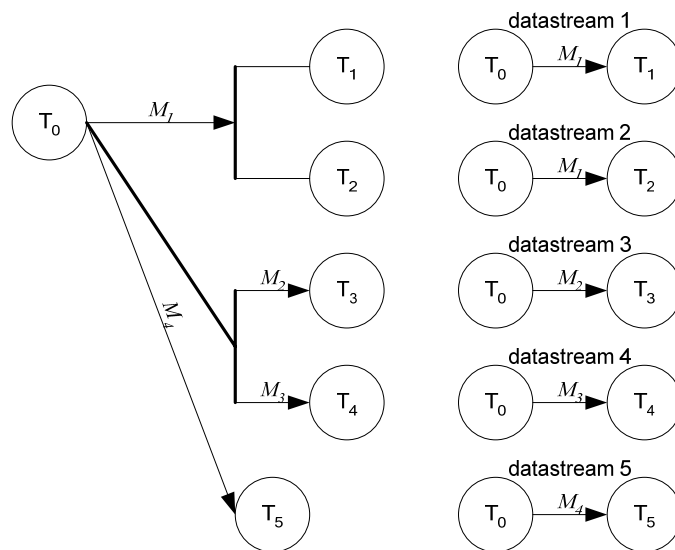
Figure 4-11 – Scenario of task interaction and corresponding data streams

In this example, five data streams were identified. Each of the data streams represents an information flow that goes from task T_0 to a consumer task.

In more complex systems, the task interaction scenarios can result in interdependent data streams. Interdependence between data streams occur when a task participates in more than one data stream. In this case, the task's calculated initial phase, *Ph*, in each data stream, depends upon the accumulated time of the previous entities, tasks and messages, and these are, most probably, different. In the rare case where the *Ph* for all data streams have the same value then it becomes the preliminary value for the task and the *Ph* of other entities of the data stream does not need to be re-evaluated. If there are at least two different values for the *Ph* then the various data streams where the task is involved have to be analysed. From this analysis, the preliminary value for the *Ph* of the task is derived. With this value, the data streams where the task participates are re-evaluated and the preliminary *Ph* values of other tasks might change.

In order to consider the interdependencies between data streams, the initial phases of each task that participates in different data streams has to be adjusted to the largest value defined by each data stream. Considering a task that participates in n data streams:

- $Ph = MAX(Ph_i)$, $i = 0, \dots , n$

This way, a task is only executed when all consumed messages are ready. The messages that arrived earlier must be kept in buffers until the consumer task is ready to consume them.

After all data streams have been re-evaluated the final *Ph* values of all tasks become the definitive values.

From the point of view of the entity, data streams can be triggered either with a task or a message. While from the point of view of time, these possibilities can be further divided into synchronous or asynchronous triggering.

A task triggering is accomplished by a producer task. On the other hand, a message triggering is accomplished by a message that is produced through unspecified means. This message can be produced by a task not controlled by the data stream analysis, or it can come from an external node, or it can result from an event generated at some node.

A synchronous task, or message, triggering is accomplished through a periodic task, or message. The period of this entity becomes the period of the data stream, T_{DS}. While an asynchronous task, or message, triggering is accomplished by an aperiodic task, or message. If aperiodic triggering can be transformed into a sporadic triggering using a sporadic server [SSL89] then a minimum inter-arrival time (MIT) can be defined. If the MIT is less then the period of the data stream then some kind of buffering technique might be needed.

4.1.3 Closed loops

For the initial phase determination process to work there is one condition that must be guaranteed, which is the inexistence of closed loops where two, or more, tasks can communicate with each other in an unbounded manner. This interdependence between tasks where two, or more, tasks change their role from producer to consumer, and vice-versa, can lead to an unbounded analysis time.

A way to provide the guarantee of inexistence of closed loops is through the use of a technique based on an ordered task list. Each element of the list is a task. After each task,

the list can not have other tasks that produce messages to the first. And before each task, the list can not have other tasks that consume messages from the first. This list is created in a way that a new task appended to the list can not produce messages to tasks that are already in the list unless it can be inserted before the consumer task without compromising the general principle. If it is possible to build an ordered task list with all tasks then the inexistence of closed loops is guaranteed.

This technique is successfully in use to avoid deadlocks in the process scheduling. It can be found on general literature that covers this topic [SGG04]. An example of a deadlock in scheduling is when several processes try to access several resources and need to have exclusive access to them. A deadlock might occur because some processes might obtain exclusive access to some resources, but not all, and be prevented from getting the remaining necessary resources. Therefore, all the involved processes will continue to wait indefinitely.

If a closed loop is found in a set of data streams then the particular task, or tasks, that are responsible for it must be divided in a way that their functionality gets separated between sub-tasks. This way the closed loop can disappear. This aspect can not be automated and is the architect that has to make the task partitioning based on the information given by the data streams.

4.1.4 Task role changing

Closed loops occur whenever a task changes its role towards other tasks with which it communicates. For example, the client-server model has two protocols that are the request-reply and the request-reply-confirmation. These protocols involve two tasks, T_0 and T_1. The request-reply involves the exchange of two messages while the latest involves the exchange of three messages.
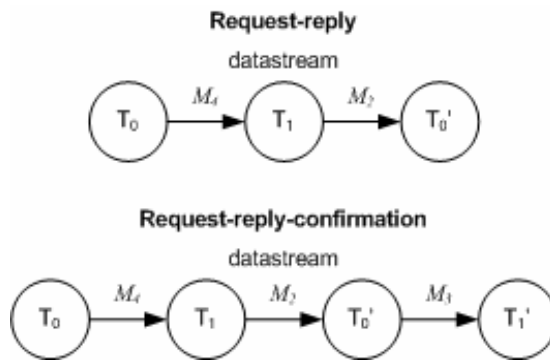
Figure 4-12 – Example of task role changing

Task T_0 begins as a producer task and then changes to a consumer task, or a consumer/producer task, according to the protocol. This means that T_0 has two functional parts. In the first part it behaves like a producer task and in the last part it behaves as either a consumer task or a consumer/producer task. The correspondent data streams are shown in Figure 4-12.

From the data streams it is clear that T_0 and T_0' are two different tasks. The same happens with T_1 and T_1'.

If tasks can be decomposed to a level where their roles do not change then closed loops are avoided.

4.1.5 Case study: HTTP-based client-server communication

HTTP defines how messages are to be formatted and what actions are to be taken by web servers and browsers in response to certain actions. HTTP is stateless, meaning that each action is executed without any knowledge of actions that came before it.
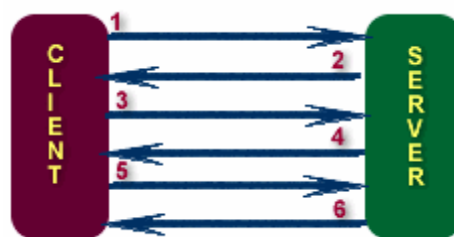
Figure 4-13 – Message exchange for a basic HTTP-based client-server communication

HTTP works in the following way [CDK05]:

1. Client requests a connection to the server
2. Server acknowledges and tells client what socket to connect on.

3. Client requests data from server.

4. Server sends requested data (or error message, etc.) to client.

5. Client acknowledges receipt of response.

6. Server closes connection.

Consider a client task and a server task that exchange the previous six messages for a basic HTTP-based communication as represented in Figure 4-13. On this scenario, the use of the interdependence analysis previously presented that checks for the existence of closed loops aborted because it is not possible to build an ordered task list that represents all the interactions between the tasks. Here follows a step-by-step description of the algorithm use:

1. The list is empty and task *client* is appended to the list;

2. Task *server* can not be appended to the list because it produces a message to task *client* that is already in the list;

3. Task *server* can not be inserted at the beginning of the list because it consumes a message from task *client* that is already in the list;

4. The interdependence analysis aborts because task *server* can not be inserted in the list.

The reason for this result is that both tasks change their role during the transaction. The client task begins as a producer task, then changes to a consumer-producer task and finally it changes to a consumer task. If this task is divided in four tasks separating clearly the different roles that were identified then the result is as shown in the data stream represented in Figure 4-14. The same rationale is used for the server task.
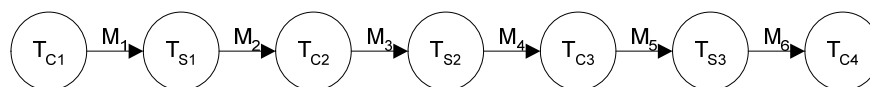
Figure 4-14 – Data stream where the client and server tasks are divided in several tasks

With this approach, now each client task has a fixed message to be consumed and/or produced.

Now supposing that the client request requires the server to get information from a sensor and that this information is acquired and processed periodically, then the new data stream representation is the one in Figure 4-15. Two new tasks are considered, these are Acquisition and Processing.
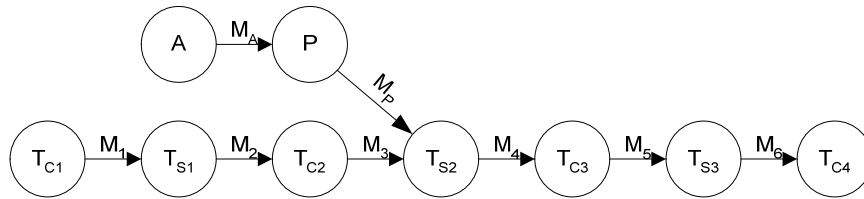
Figure 4-15 – Resulting data streams

In this scenario, two data streams were identified. Task T_{S2} participates in both data streams. Using the algorithm previously presented that checks for the existence of closed loops results in an ordered task list with every tasks. Therefore, the conclusion is that there are no closed loops because the ordered task list includes all the tasks.

4.2 Information flow control

The information flow of the process can be controlled offline or online. The offline control implies that all decisions have to be taken before the system starts to function. In this situation the system cannot be changed, i.e. neither can the parameters of the entities, tasks or messages, be changed nor can the number of entities be changed. The online control combines the parameter setup before the system starts to function with an adaptive run-time system adjustment. Naturally, this flexibility comes at the cost of an additional complexity. If this complexity can be bounded and be implemented within an acceptable cost frame, for a process that needs continuous control, the online control is superior to the offline control. Even if the process to be controlled is simple, adjustments are always required after some time. In an offline control, the adjustment requires the system to be stopped, reconfigured and restarted. This cost should not be forgotten when evaluating the type of approach to be taken.

From the point of view of the system architecture, a simple offline control of the process requires that the set of tasks' and messages' parameters is previously loaded at each node. This way, each node will always know what to do. But this might bring a serious drawback in the case where the nodes have low computational power. Due to the overhead in the form of processing and also in the form of extra storage space, another approach can be considered. This approach consists in a special node that manages the tasks' and messages' parameters. This special node would then inform each node about the tasks to be carried out. Now, independently of the approach taken, there is the need to synchronize the nodes in their access to the network.

48

On the other hand, to have an online control, either the tasks' and messages' parameters have to be updatable at each node or, when considering the use of a special node, the approach is similar to the offline control.

In order to control the information flows, the timing aspects of the tasks and messages have to be taken into account. Therefore, a scheduling that considers both tasks and messages in a holistic approach is required.

In order to produce a schedule, two phases can be contemplated that are parameter determination and scheduling.

4.3 Holistic scheduling

In this study, the parameter determination and tuning, and the scheduling are centralized, online and holistic. A centralized parameter determination and scheduling can be accomplished in any regular node of the system or in a dedicated node, depending on the available resources and policy. They are also holistic because they encompass all tasks and messages.

The task interactions previously presented are now used to support the parameter determination and tuning of tasks and messages towards a holistic scheduling. The parameter determination is based on new scheduling options like the most constrained resource, either the network or the nodes, and the relation between the execution and transmission windows of related tasks and messages. This results in three types of approaches that are, respectively, network-centric, node-centric and overlapping. The first two approaches can also be designated as non-overlapping approaches.

The execution window of a producer task may overlap, or not, the transmission window of the corresponding produced message. The same can be said about the transmission window of a message and the execution window of the corresponding consumer task. If these windows do not overlap, then the release instant of an entity is not dependent of the release instant of the previous entity. The net-centric and the node-centric approaches have non-overlapping windows, while the third approach may have overlapping windows. The proposed techniques are to be used before the actual scheduling of both tasks and messages, even though some pre-scheduling can be used in some techniques.

Assumptions

For this study the following assumptions are considered:

- Tasks are periodic and interactive.

- Tasks have a fixed computation time, or at least a fixed upper bound on their computation times, which is less than or equal to their period.

- No task may voluntarily suspend itself.

- Tasks are fully preemptive.

- Tasks are released at the beginning of each period.

- Messages are consumed at the beginning of their execution and are produced at the end of their execution.

- Tasks can have precedence relationships.

- The periods and the initial phases have to be multiple of a pre-defined time value, the elementary cycle (EC). This is because there is no control of the task and message dispatching between two consecutive trigger messages as it is the case of CAN based FTT systems.

- Any two entities, tasks or messages, with a precedence relation occur in different ECs. The reason for this restriction is that it is not possible to control the dispatching inside an EC which defines the time resolution.

- An interval with a pattern, of tasks and messages, that will be repeated indefinitely, until one of the tasks finishes or an error occurs, is called a Macro-cycle.

General parameter definition

Some tasks and messages parameters have to be defined at creation time, namely:

- Stand-alone tasks – C, T, D, Pr, MP and MC;

- Interactive tasks – C, Pr, MP and MC;

- Messages – C, Pr, PT and CTL.

The node of execution of a task, parameter N_i, might not be specified by the user, meaning that the Master is free to allocate it to the best Station. The definition of the best Station is beyond the scope of this work, but several aspects can be considered like: load balancing, communications performance, availability, usage of unique hardware or software capabilities, etc… All the other parameters (of tasks and messages) are determined during analysis phase.

4.3.1 Non-overlapping approaches

When the execution window is independent of the transmission window of a related message this means that the release instant of a task is independent of the release instant of the related message.

The parameter determination of tasks and messages can be a stand-alone process or may require the use of intermediate scheduling. These two techniques are called, respectively, independent scheduling and dependent scheduling.

There can be two approaches to the parameter determination of tasks and messages, namely:

- A <u>net-centric</u> approach, where messages impose restrictions to the set of tasks, and
- A <u>node-centric</u> approach, where tasks impose restrictions to the set of messages.

These approaches were first proposed in [CF02] and [CF04], and are considered for the parameter determination. A more general view of the problem leads us to analyse in which situations each of the approaches should be used. The assessment factor can be the weight of use of the system resources, the network and the nodes. Four combinations can then occur:

- <u>Low</u> network load and <u>low</u> node load, where any approach can be taken;
- <u>High</u> network load and <u>low</u> node load, corresponds to the net-centric approach;
- <u>Low</u> network load and <u>high</u> node load, corresponds to the node-centric approach;
- <u>High</u> network load and <u>high</u> node load, where both approaches should be considered in order to select one.

The space of combinations is depicted in Figure 4-16.
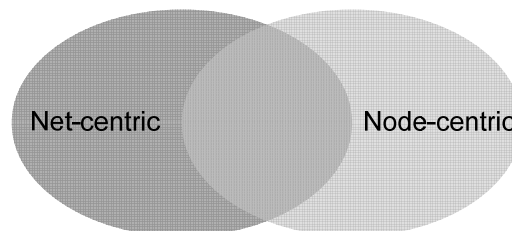
Figure 4-16 – Combinations of net-centric and node-centric approaches

Considering the basic interactions between tasks, the parameter determination of tasks and messages will now be presented according to the two previous approaches.

4.3.2 Net-centric approaches

A net-centric approach is more suited to network resource constrained environments. Such an approach tries to push the constraints to the node resources leaving the message requirements as relaxed as possible. In the following sections, three net-centric approaches are presented.

In order to achieve a holistic schedule of tasks and messages several steps have to be done. Firstly, the precedence graphs are built. Using the data flow information from these graphs, a preliminary Ph for messages is derived (without considering the tasks) and the Macro-cycle is calculated.

After this, scheduling takes place. Any scheduling algorithm can be used and its choice is beyond the scope of this work. There are two approaches to the task and message scheduling, namely: independent scheduling and dependent scheduling. In dependence of the approach selected, scheduling can be done simultaneously for tasks and messages (independent scheduling) or message scheduling is done before task scheduling (dependent scheduling).

Dependent scheduling is a process with two phases. In phase one, messages are scheduled and new message parameters, like the jitter, can be obtained. Using these new message parameters, a set of parameters is derived for each task in every precedence graph. Finally, in phase two, tasks are scheduled.

If the scheduler is unable to build a Macro-cycle due to a task deadline miss, a reverse parameter deriving can be tried. In this process the task that missed the deadline will have its parameters relaxed resulting in an increase of constraints of the produced and/or consumed messages. This method resembles the node-centric approach for a particular task, where tasks impose restrictions to the messages.

4.3.2.1   General parameter restrictions

In order to define the parameter restrictions it is assumed that these are message-based, which means that the messages impose restrictions to the tasks. Due to this, all messages have to be fully specified, apart from their initial phase (Ph).

In Chapter 2, a nomenclature for the definition of the parameters of tasks and messages was defined. Considering this nomenclature and the general parameter definition in section 4.3, the specific parameters that have to be defined at creation time are:

- Messages – T and D.

In order to generate the restrictions for the tasks, it is necessary to define the equations that permit the calculus of the tasks' parameters based on the messages' parameters.

Interactive tasks have restrictions in dependence of the particular approach presented in the following sections, while stand-alone tasks have no restrictions.

4.3.2.2 Scheduling independent approach: message deadline

In this approach, tasks are started considering that the messages may be transmitted anytime, in the transmission window, up to its deadline. Also tasks may be executed anytime in their execution window. Its characterization follows in the next sections.

Parameter restrictions

According to the type of interactive task, several restrictions apply in dependence of the messages produced and/or consumed, namely:
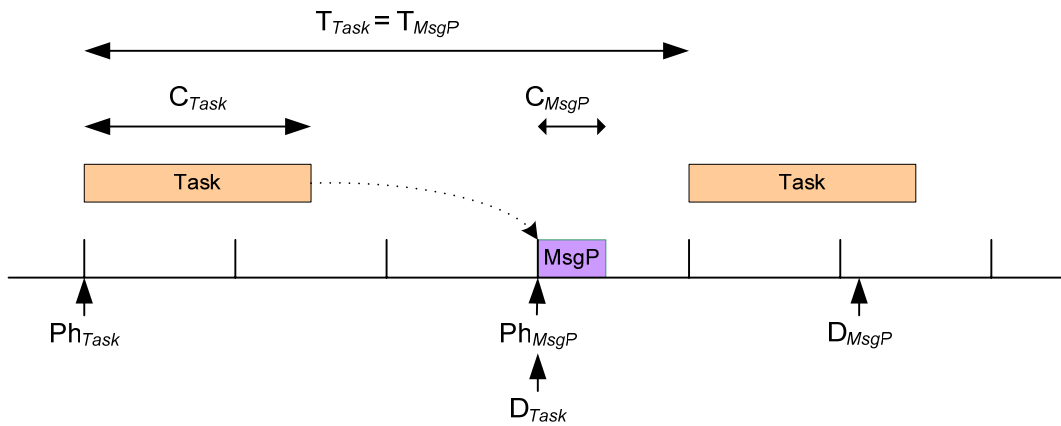
- **Producer tasks**

Figure 4-17 – MD: Task produces a message

Figure 4-17 shows a producer task *Task* that produces a message *MsgP*. The general restrictions for a producer task are:

$$T_{Task} = T_{MsgP} = T_{DS}$$

(4-1)

$$Ph_{Task} < Ph_{MsgP}$$

(4-2)

53

Furthermore,

$$r_{Task,j} < r_{MsgP,j} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-3)</div>

In the previous equation, $r_{Task,j}$ is the release instant of the instance j of task *Task*, $r_{MsgP,j}$ is the release instant of the instance j of message *MsgP* and *nInst* is the number of instances of both *Task* and *MsgP*. This means that, for any instance of *Task* and *MsgP*, the release instant of *Task* is always inferior to the release instant of *MsgP*. If a producer task is always the first entity in a data stream then:

$$r_{Task,1} = 0$$

<div align="right">(4-4)</div>

Other restrictions for a producer task are:

$$r_{Task,j+1} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \geq d_{MsgP,j} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-5)</div>

$$d_{MsgP,j} = r_{MsgP,j} + D_{MsgP} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-6)</div>

$$r_{MsgP,j} = r_{Task,j} + \left\lceil \frac{D_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-7)</div>

Because a produced message can only be transmitted in the EC after the deadline, in order to maximize the task's execution window, D_{Task} is always a multiple of the t_{EC}. Therefore equation (4-7) becomes:

$$r_{MsgP,j} = r_{Task,j} + D_{Task} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-8)</div>

From equations (4-5), (4-6) and (4-8), the lowest value for $r_{Task,j+1}$ that is a multiple of t_{EC} becomes:

$$r_{Task,j+1} = \left\lceil \frac{d_{MsgP,j}}{t_{EC}} \right\rceil \times t_{EC} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$r_{Task,j+1} = \left\lceil \frac{r_{MsgP,j} + D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$r_{Task,j+1} = \left\lceil \frac{r_{Task,j} + \left\lceil \frac{D_{Task}}{t_{EC}} \right\rceil \times t_{EC} + D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$r_{Task,j} + T_{DS} = r_{Task,j} + \left\lceil \frac{D_{Task}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$\left\lceil \frac{D_{Task}}{t_{EC}} \right\rceil \times t_{EC} = T_{DS} - \left\lceil \frac{D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC}$$

Since D_{Task} is naturally a multiple of t_{EC}, the previous equation becomes:

$$D_{Task} = T_{DS} - \left\lceil \frac{D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-9)

This guarantees that, when the task finishes and a message is produced, the previous message has already been transmitted. It can be understood from this equation that, when the D_{MsgP} comes closer to the value of T_{DS}, the D_{Task} approaches the value of C_{Task}. So we can conclude that, when the transmission window of the message widens, the execution window of the task narrows, and vice-versa.

Having the task's deadline calculated and in accordance with equation (4-2) a new restriction can be obtained from:

$$Ph_{Msg} \geq Ph_{Task} + D_{Task}$$

(4-10)

$$r_{MsgP,j} \geq r_{Task,j} + D_{Task} \qquad \forall j = 1,...,nInst$$

(4-11)

This guarantees that, when the task's deadline is reached, the message to be transmitted has already been produced.

Considering a producer task *Task* (represented by T in Figure 4-17) that produces a message *MsgP* (represented by P in Figure 4-17), from equation (4-10), and maximizing the execution window of the task, the message's initial phase, Ph_{MsgP}, is derived as follows:

$$Ph_{MsgP} = Ph_{Task} + D_{Task}$$

(4-12)

- **Consumer tasks**
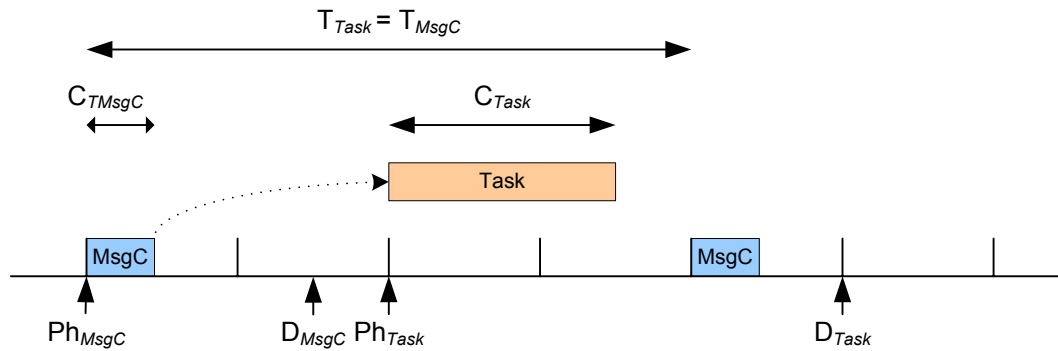
Figure 4-18 – MD: Task consumes a message

Figure 4-18 shows a consumer task *Task* that consumes a message *MsgC*. The general restrictions for a consumer task are:

$$T_{Task} = T_{MsgC} = T_{DS}$$

(4-13)

$$Ph_{MsgC} < Ph_{Task}$$

(4-14)

Furthermore,

$$r_{MsgC(j)} < r_{Task(j)} \qquad \forall j = 1,...,nInst$$

(4-15)

Another restriction for a consumer task is:

$$r_{Task,j} \geq d_{MsgC,j} \qquad \forall j = 1,...,nInst$$

(4-16)

This guarantees that, when the task starts, the message to be consumed has already been transmitted.

56

$$d_{MsgC,j} = r_{MsgC,j} + D_{MsgC} \qquad\qquad \forall j = 1,...,nInst$$

<div align="right">(4-17)</div>

$$r_{MsgC,j} \geq d_{Task,j-1} - C_{Task} \qquad\qquad \forall j = 1,...,nInst$$

<div align="right">(4-18)</div>

From equation (4-18), the minimum value for $r_{MsgC,j}$ becomes:

$$r_{MsgC,j} = d_{Task,j-1} - C_{Task} \qquad\qquad \forall j = 1,...,nInst$$

<div align="right">(4-19)</div>

$$r_{MsgC,j} = r_{Task,j-1} + D_{Task} - C_{Task} \qquad\qquad \forall j = 1,...,nInst$$

<div align="right">(4-20)</div>

From this equation, it can be understood that D_{Task}-C_{Task} is a multiple of t_{EC}, even though, neither D_{Task} or C_{Task} have to be multiples of t_{EC}.

Considering that a message is only available to be consumed at the beginning of the EC, following its transmission, in order to maximize D_{Task}, the previous equation can be rewritten as:

$$r_{MsgC,j} = r_{Task,j-1} + D_{Task} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad\qquad \forall j = 1,...,nInst$$

<div align="right">(4-21)</div>

From equation (4-16), the minimum value for $r_{Task,j}$ becomes:

$$r_{Task,j} = \left\lceil \frac{d_{MsgC,j}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-22)</div>

From this equation, and using equations (4-17) and (4-20), D_{Task} can be derived as follows:

$$r_{Task,j} = \left\lceil \frac{r_{MsgC,j} + D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$r_{Task,j} = \left\lceil \frac{d_{Task,j-1} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} + D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$r_{Task,j} = \left\lceil \frac{r_{Task,j-1} + D_{Task} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} + D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$r_{Task,j} = r_{Task,j} - T_{DS} + \left\lceil \frac{D_{Task} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} + D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

$$T_{DS} = D_{Task} - \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

$$D_{Task} = T_{DS} - \left\lceil \frac{D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-23)

This guarantees that, when the task starts, the next message to be consumed has not been transmitted yet. It can be understood from this equation that, when D_{MsgC} comes closer to the value of T_{DS}, D_{Task} approaches the value of C_{Task}. So we can conclude that, when the transmission window of the message widens, the execution window of the task narrows, and vice-versa.

Maximizing the execution window of *Task*, the task's initial phase, Ph_{Task} is derived as follows:

$$Ph_{Task} = Ph_{MsgC} + \left\lceil \frac{D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-24)

- ## Consumer/producer tasks
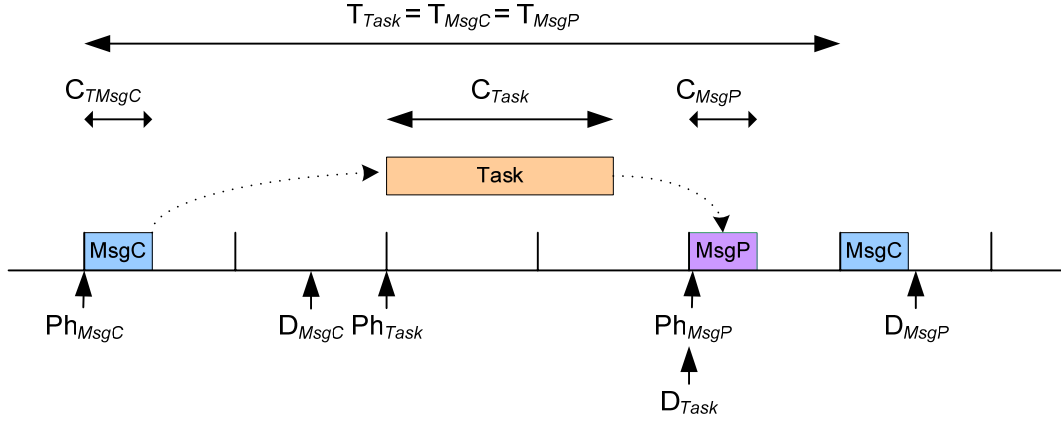
Figure 4-19 – MD: Task consumes and produces a message

Figure 4-19 shows a consumer/producer task *Task* that consumes a message *MsgC* and produces a message *MsgP*. The general restrictions for a consumer/producer task are:

$$T_{Task} = T_{MsgC} = T_{MsgP} = T_{DS}$$

(4-25)

$$Ph_{MsgC} < Ph_{Task} < Ph_{MsgP}$$

(4-26)

Furthermore,

$$r_{MsgC,j} < r_{Task,j} < r_{MsgP,j} \qquad \forall j = 1,...,nInst$$

(4-27)

Another restriction for a consumer/producer task was already presented in equation (4-15). Considering equations (4-9) and (4-23), the task's deadline is given by:

$$D_{Task} \leq MIN(D_{TaskP}, D_{TaskC})$$

(4-28)

Where D_{TaskP} is the deadline of a producer task and D_{TaskC} is the deadline of a consumer task.

Maximizing the execution window of the task, D_{Task} is finally derived as:

$$D_{Task} = MIN\left(T_{DS} - \left\lceil \frac{D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC}, T_{DS} - \left\lceil \frac{D_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \right)$$

(4-29)

The Ph_{Task} can be determined with the equation (4-24) and the Ph_{MsgP} can be calculated with the following equation:

$$Ph_{MsgP} = Ph_{Task} + \left\lceil \frac{D_{Task}}{t_{EC}} \right\rceil \times t_{EC}$$

The task release instant (Ph_{Task} in the case of the first release), must occur in the EC after the deadline of the message being consumed (D_{MsgC}). This guarantees that the message has been transmitted and the data to be consumed has already been delivered to the kernel when the EC trigger message, that dispatches the task, arrives.

The task deadline (D_{Task}), which is measured relatively to the task release instant, must occur in the EC before both the release instant of the message being produced (Ph_{MsgP} in the case of the first release) and the instant that would make possible the next message C to be consumed by the current task.

It can be understood from these equations that, depending of the deadlines of the produced and consumed messages, the message with the largest transmission window will prevail when defining the execution window of the task.

If $D_{TaskP} > D_{TaskC}$ then the difference can be used to increase the D_{MsgP} (while reducing the Ph_{MsgP}) relaxing the produced message constraints. So the final value would become:

$$D_{MsgP} = D_{MsgP} + \left\lfloor \frac{D_{TaskP} - D_{TaskC}}{t_{EC}} \right\rfloor \times t_{EC}$$

On the contrary, if $D_{TaskP} < D_{TaskC}$ then the difference can be used to increase the D_{MsgC}, relaxing the consumed message constraints. So the final value would become:

$$D_{MsgC} = D_{MsgC} + \left\lfloor \frac{D_{TaskC} - D_{TaskP}}{t_{EC}} \right\rfloor \times t_{EC}$$

This new value leads to the re-evaluation of Ph_{Task} calculated in equation (4-24).

4.3.2.3   Scheduling dependent approach: message maximum finishing

This approach is implemented in two phases. In phase one, messages are scheduled to be transmitted anytime in the transmission window, and the maximum finishing time for each message, $MAXf_{Msg}$, in the Macro-cycle is determined. This maximum finishing time is

calculated upon the maximum jitter for each message in a complete Macro-cycle. The finishing time for each message is calculated upon the message jitter, J_{Msg}, and its finishing time, C_{Msg}, with the following equation:

$$f_{Msg} = J_{Msg} + C_{Msg}$$

(4-33)

As previously presented, each synchronous message m, SM_m, is defined in the Macro-cycle as the set of instances, $SM_{m,j}$ where $j=1,...,nInst$. The number of instances of a particular synchronous message m in the interval of a Macro-cycle, t_{MC}, is given by:

$$nInst = \frac{t_{MC}}{T_m}$$

(4-34)

The maximum finishing time is calculated in dependence of the maximum jitter, which is calculated upon the jitter of each instance of a message, $J_{Msg,j}$, with the following equation:

$$MAXf_{Msg} = MAX\left(f_{Msg,j}\right) = MAX\left(J_{Msg,j}\right) + C_{Msg}$$

(4-35)

In phase two, tasks are scheduled to be executed anytime in their execution window.

Parameter restrictions

According to the type of interactive task, several restrictions apply in dependence of the messages produced and/or consumed, namely:
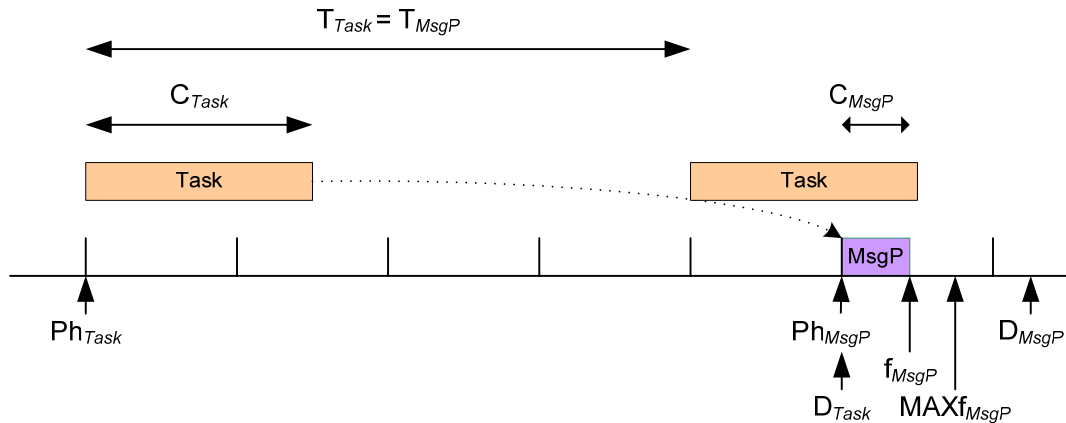
- **Producer tasks**

Figure 4-20 – MMF: Task produces a message

61

Figure 4-20 shows a producer task *Task* that produces a message *MsgP*. A general restriction for a producer task is:

$$T_{Task} = T_{MsgP} = T_{DS}$$

<div align="right">(4-36)</div>

From equation (4-9) and considering the *MAXf* of a message instead of its deadline, the task's deadline is derived as follows:

$$D_{Task} = T_{DS} - \left\lceil \frac{MAXf_{MsgP} - C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

<div align="right">(4-37)</div>

And the message's initial phase is given by:

$$Ph_{MsgP} = Ph_{Task} + D_{Task}$$

<div align="right">(4-38)</div>

- **Consumer tasks**
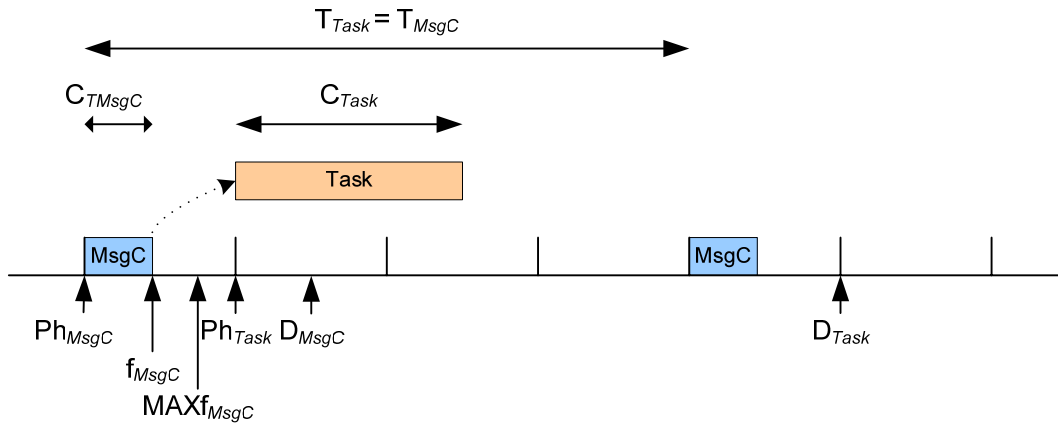
Figure 4-21 – MMF: Task consumes a message

Figure 4-21 shows a consumer task *Task* that consumes a message *MsgC*. A general restriction for a consumer task is:

$$T_{Task} = T_{MsgC} = T_{DS}$$

<div align="right">(4-39)</div>

62

The task's initial phase is given by:

$$Ph_{Task} = Ph_{MsgC} + \left\lceil \frac{MAXf_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

<div align="right">(4-40)</div>

From equation (4-9) and considering the *MaxTr* of a message instead of the deadline, the task's deadline is derived as follows:

$$D_{Task} = T_{DS} + C_{Task} - \left\lceil \frac{MAXf_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

<div align="right">(4-41)</div>

- **Consumer/producer tasks**
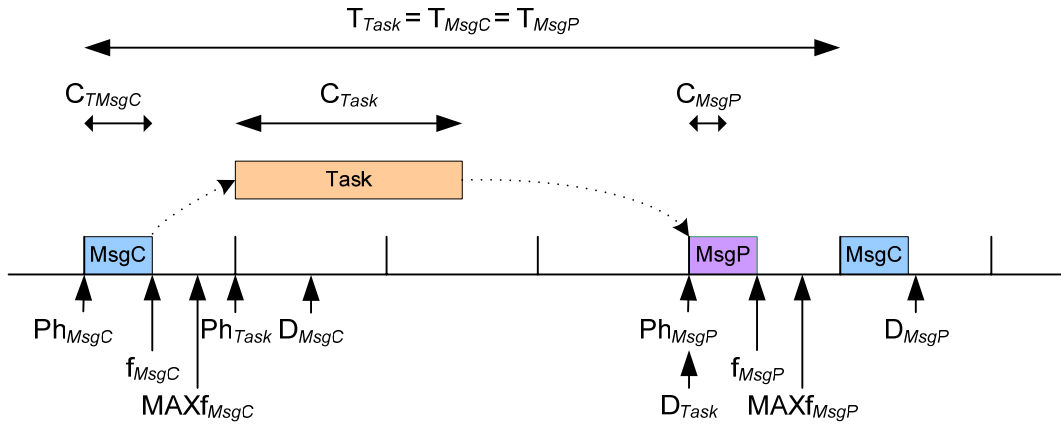
Figure 4-22 – MMF: Task consumes and produces a message

Figure 4-22 shows a consumer/producer task *Task* that consumes a message *MsgC* and produces a message *MsgP*. A general restriction for consumer/producer tasks is:

$$T_{Task} = T_{MsgC} = T_{MsgP} = T_{DS}$$

<div align="right">(4-42)</div>

From equation (4-29) and considering the *MAXf* of a message instead of its deadline, the task's deadline is derived as follows:

$$D_{Task} = MIN\left(T_{DS} - \left\lceil \frac{MAXf_{MsgP} - C_{Task}}{t_{EC}} \right\rceil \times t_{EC}, T_{DS} + C_{Task} - \left\lceil \frac{MAXf_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} \right)$$

<div align="right">(4-43)</div>

Ph_{Task} can be determined with the equation (4-40) and Ph_{MsgP} can be calculated with the equation (4-30).

As in section 4.3.2.2 (consumer/producer tasks), this process could lead to the re-evaluation of some parameters.

4.3.2.4 Scheduling dependent approach: message finishing

This approach is implemented again in two phases. In phase one, messages are scheduled and the finishing time, f_{Msg}, for each instance of each message in the Macro-cycle is determined. In phase two, tasks are scheduled to be dispatched according to the finishing time of each instance of each message.

Parameter restrictions

Due to the similarity with the message maximum finishing approach, this section only shows the final equations. The reasoning behind is similar to the previous approach but different parameters result from each message instance. So this means that instead of calculating a single set of tasks' and messages' parameters for the whole macro-cycle, in this approach a separate set of parameters has to be calculated for each task's and message's instances. Namely a finishing instant, f_{Msg}, for each message, has to be determined.

According to the type of interactive task, several restrictions apply in dependence of the messages produced and/or consumed, namely:

- **Producer tasks**

Considering a producer task *Task* that produces a message *MsgP* (see Figure 4-20), maximizing the execution window of *Task*, the task's period, T_{Task}, the task's deadline, D_{Task}, and the message's initial phase, Ph_{MsgP}, are derived as follows:

$$T_{Task} = T_{MsgP} = T_{DS}$$

(4-44)

$$D_{Task} = T_{DS} - \left\lceil \frac{f_{MsgP} - C_{Task}}{t_{EC}} \right\rceil \times t_{EC} \qquad \forall j = 1,...,nInst$$

(4-45)

64

$$Ph_{MsgP} = Ph_{Task} + D_{Task}$$

<div align="right">(4-46)</div>

- **Consumer tasks**

Considering a consumer task *Task* that consumes a message *MsgC* (see Figure 4-21), maximizing the execution window of *Task*, the task's period, T_{Task}, the task's initial phase, Ph_{Task}, and the task's deadline, D_{Task}, are derived as follows:

$$T_{Task} = T_{MsgC} = T_{DS}$$

<div align="right">(4-47)</div>

$$Ph_{Task} = Ph_{MsgC} + \left\lceil \frac{f_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

<div align="right">(4-48)</div>

$$D_{Task} = T_{DS} + C_{Task} - \left\lceil \frac{f_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

<div align="right">(4-49)</div>

- **Consumer/producer tasks**

Considering a consumer/producer task *Task* that consumes a message *MsgC* and produces a message *MsgP* (see Figure 4-22), maximizing the execution window of *Task*, the task's period, T_{Task}, and the task's deadline, D_{Task}, are derived as follows:

$$T_{Task} = T_{MsgC} = T_{MsgP} = T_{DS}$$

<div align="right">(4-50)</div>

$$D_{Task} = MIN\left(T_{DS} - \left\lceil \frac{f_{MsgP} - C_{Task}}{t_{EC}} \right\rceil \times t_{EC}, T_{DS} + C_{Task} - \left\lceil \frac{f_{MsgC}}{t_{EC}} \right\rceil \times t_{EC} \right)$$

<div align="right">(4-51)</div>

The Ph_{Task} can be determined with the equation (4-48) and the Ph_{MsgP} can be calculated with the equation (4-46).

4.3.2.5 Comparing the 3 scheduling approaches

The message deadline approach is independent of the message jitter and makes the consumer tasks periodic. On the other hand, due to the reliance on the worst case transmission time this approach leads to the latest release time for tasks that consume a message. This side effect reduces the execution window of these tasks to the minimum and is therefore non-optimal.

Although the message maximum transmission approach is dependent of the message jitter, it still makes the consumer tasks periodic because the maximum jitter of every message is calculated for the macro-cycle. This approach also produces the earliest release time for periodic tasks.

These statements can be easily verified because:

$$MAXf_{MsgP} \leq D_{MsgP}$$

(4-52)

$$MAXf_{MsgC} \leq D_{MsgC}$$

(4-53)

This leads to a D_{Task} in the message maximum transmission approach, which is higher or equal than the corresponding D_{Task} in the message deadline approach.

The message transmission approach leads to the earliest release times for tasks. But on the other hand, it introduces jitter in the tasks due to different message jitters. This side effect could be undesirable for many systems.

4.3.3 Node-centric approaches

On environments with high node resource constraints and low network resource constraints, the parameter determination technique can be reversed. This way tasks impose restrictions to the message set.

This approach is the opposite of the net-centric approach, where tasks have to be fully specified, apart from Ph, and impose restrictions to the messages.

A new set of equations can be derived for the various approaches leading to the full specification of tasks and messages. In the following sections, three node-centric approaches are presented.

In section 4.3.2, independent and dependent scheduling techniques were introduced for the net-centric approaches. These techniques can also be used for the node-centric approaches while keeping in mind that, in this case, the tasks impose restrictions to the messages.

4.3.3.1 General parameter restrictions

In order to define the parameter restrictions it is assumed that these are task-based, which means that the tasks impose restrictions to the messages. Due to this, all tasks have to be fully specified, apart from their initial phase (Ph).

In Chapter 2, a nomenclature for the definition of the parameters of tasks and messages was defined. Considering this nomenclature and the general parameter definition in section 4.3, the specific parameters that have to be defined at creation time are:

- Interactive tasks – T and D.

In order to generate the restrictions for the tasks, it is necessary to define the equations that permit the calculus of the tasks' parameters based on the messages' parameters.

Interactive tasks have restrictions in dependence of the particular approach presented in the following sections, while stand-alone tasks have no restrictions.

4.3.3.2 Scheduling independent approach: task deadline

In this approach, messages are started considering that the tasks may be executed anytime, in the transmission window, up to its deadline. Also messages may be transmitted anytime in their transmission window. Its characterization follows.

Parameter restrictions
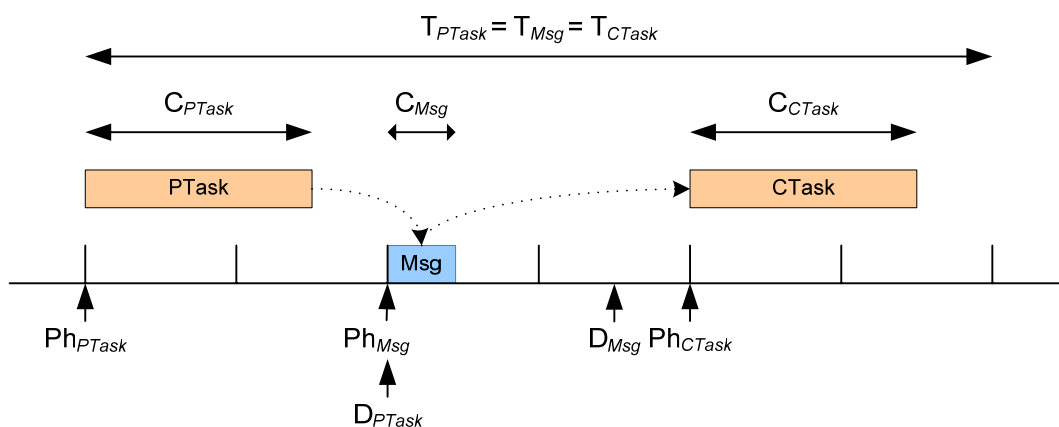
Figure 4-23 – TD: Message that is produced and consumed

A message *Msg* (see Figure 4-23) is produced by task *PTask* and is consumed by task *CTask*. The message's period, T_{Msg}, is equal to the periods of *PTask* and *CTask*.

The message's initial phase, Ph_{Msg}, is defined upon the producer task characteristics and is given by:

$$Ph_{Msg} = Ph_{PTask} + \left\lceil \frac{D_{PTask}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-54)

Considering equation (4-9) and that D_{Msg}-C_{CTask} is a multiple of t_{EC}, the restriction imposed by *PTask* to the produced message's deadline is:

$$D_{MsgP} = T_{DS} - D_{PTask} + C_{PTask}$$

(4-55)

On the other hand, considering equation (4-23), the restriction imposed by *CTask* to the consumed message's deadline is:

$$D_{MsgC} = T_{DS} + \left\lfloor \frac{C_{CTask} - D_{CTask}}{t_{EC}} \right\rfloor \times t_{EC}$$

(4-56)

From the previous equations the message deadline is given by:

$$D_{Msg} \leq MIN\left(D_{MsgP}, D_{MsgC}\right)$$

(4-57)

Maximizing the transmission window of the message, D_{Msg} is finally derived as:

$$D_{Msg} = MIN\left(T_{DS} - C_{PTask} - \left\lceil \frac{D_{PTask}}{t_{EC}} \right\rceil \times t_{EC}, T_{DS} + \left\lfloor \frac{C_{CTask} - D_{CTask}}{t_{EC}} \right\rfloor \times t_{EC} \right)$$

(4-58)

4.3.3.3 Scheduling dependent approach: task maximum finishing

This approach is implemented in two phases. In phase one, tasks are scheduled to be executed anytime in the execution window, and the maximum finishing time for each task, Maxf_{Task}, in the Macro-cycle is determined. This maximum finishing time is calculated upon the maximum jitter for each task in a complete macro-cycle. The finishing time for each task is calculated upon the task jitter, J_{Task}, and its execution time, C_{Task}, with the following equation:

$$f_{Task} = J_{Task} + C_{Task}$$

As previously presented, each synchronous task i, ST_i, is defined in the Macro-cycle as the set of instances, $ST_{i,j}$ where $j=1,...,nInst$. The number of instances of a particular synchronous task i in the interval of a Macro-cycle, t_{MC}, is given by:

$$nInst = \frac{t_{MC}}{T_i}$$

The maximum finishing time is calculated in dependence of the maximum jitter, which is calculated upon the jitter of each instance of a task, $J_{Task,j}$, with the following equation:

$$MAXf_{Task} = MAX(f_{Task,j}) = MAX(J_{Task,j}) + C_{Task}$$

In phase two, tasks are scheduled to be executed anytime in their execution window.

Parameter restrictions
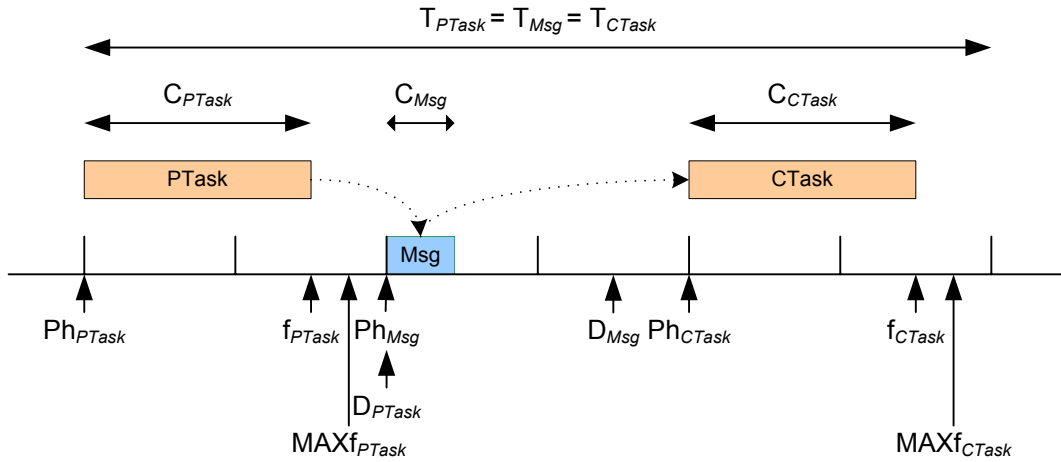
Figure 4-24 – TMF: Message that is produced and consumed

A message *Msg* (see Figure 4-24) is produced by task *PTask* and is consumed by task *CTask*. The message's period, T_{Msg}, is equal to the periods of *PTask* and *CTask*.

The message's initial phase, Ph_{Msg}, is defined upon the producer task characteristics and is given by:

$$Ph_{Msg} = Ph_{PTask} + \left\lceil \frac{MAXf_{PTask}}{t_{EC}} \right\rceil \times t_{EC}$$

Considering equation (4-9) and that $D_{Msg}-C_{CTask}$ is a multiple of t_{EC}, the restriction imposed by *PTask* to the produced message's deadline is:

$$D_{MsgP} = T_{DS} - C_{PTask} - MAXf_{PTask}$$

<div align="right">(4-63)</div>

On the other hand, considering equation (4-23), the restriction imposed by *CTask* to the consumed message's deadline is:

$$D_{MsgC} = T_{DS} + \left\lfloor \frac{C_{CTask} - MAXf_{CTask}}{t_{EC}} \right\rfloor \times t_{EC}$$

<div align="right">(4-64)</div>

From the previous equations the message deadline is given by:

$$D_{Msg} \leq MIN(D_{MsgP}, D_{MsgC})$$

<div align="right">(4-65)</div>

Maximizing the transmission window of the message, D_{Msg} is finally derived as:

$$D_{Msg} = MIN\left(T_{DS} - C_{PTask} - MAXf_{PTask}, T_{DS} + \left\lfloor \frac{C_{CTask} - MAXf_{CTask}}{t_{EC}} \right\rfloor \times t_{EC}\right)$$

<div align="right">(4-66)</div>

4.3.3.4 Scheduling dependent approach: task finishing

This approach is implemented in two phases. In phase one, tasks are scheduled and the finishing time, f_{Task}, for each instance of each task in the Macro-cycle is determined. In phase two, messages are scheduled to be dispatched according to the finishing time of each instance of each task.

Parameter restrictions

Due to the similarity with the task maximum finishing approach, this section only shows the final equations. The reasoning behind is similar to the previous approach but different parameters result from each task instance. So this means that instead of calculating a single set of tasks' and messages' parameters for the whole macro-cycle, in this approach a separate set of parameters has to be calculated for each task's and message's instances. Namely a finishing instant, f_{Task}, for each task, has to be determined.

A message *Msg* (see Figure 4-24) is produced by task *PTask* and is consumed by task *CTask*. The message's period, T_{Msg}, is equal to the periods of *PTask* and *CTask*.

The message's initial phase, Ph_{Msg}, is defined upon the producer task characteristics and is given by:

$$Ph_{Msg} = Ph_{PTask} + \left\lceil \frac{f_{PTask}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-67)

Considering equation (4-9) and that D_{Msg}-C_{CTask} is a multiple of t_{EC}, the restriction imposed by *PTask* to the produced message's deadline is:

$$D_{MsgP} = T_{DS} - C_{PTask} - f_{PTask}$$

(4-68)

On the other hand, considering equation (4-23), the restriction imposed by *CTask* to the consumed message's deadline is:

$$D_{MsgC} = T_{DS} + \left\lfloor \frac{C_{CTask} - f_{CTask}}{t_{EC}} \right\rfloor \times t_{EC}$$

(4-69)

From the previous equations the message deadline is given by:

$$D_{Msg} \leq MIN\left(D_{MsgP}, D_{MsgC}\right)$$

(4-70)

Maximizing the transmission window of the message, D_{Msg} is finally derived as:

$$D_{Msg} = MIN\left(T_{DS} - C_{PTask} - f_{PTask}, T_{DS} + \left\lfloor \frac{C_{CTask} - f_{CTask}}{t_{EC}} \right\rfloor \times t_{EC} \right)$$

(4-71)

4.3.3.5   Comparing the 3 scheduling approaches

The task deadline approach is independent of the task jitter and makes the messages periodic. On the other hand, due to the reliance on the worst case execution time this approach leads to the latest release time for messages. This side effect reduces the transmission window of these messages to the minimum and is therefore non-optimal.

Although the task maximum finishing approach is dependent of the task jitter, it still makes the messages periodic because the maximum jitter of every task is calculated for the macro-cycle. This approach also produces the earliest release time for messages.

These statements can be easily verified because:

$$MAXf_{Task} \leq D_{Task}$$

$$MAXf_{Task} \leq D_{Task}$$

This leads to a D_{Msg}, in the task maximum finishing approach, which is higher or equal than the corresponding D_{Msg}, in the task deadline approach.

The task finishing approach leads to the earliest release times for messages. But on the other hand, it introduces jitter in the messages due to different task jitters. This side effect could be undesirable for many systems.

4.3.4 Summary of the non-overlapping approaches

A summary of the two sets of approaches follows in Table 4-2 and Table 4-3.

Approach	Method	Independent windows	Task and message scheduling
Net-centric	Message Deadline (**MD**)	Yes	Independent
	Message Maximum Finishing (**MMF**)	Yes	Dependent
	Message Finishing (**MF**)	Yes	Dependent
Node-centric	Task Deadline (**TD**)	Yes	Independent
	Task Maximum Finishing (**TMF**)	Yes	Dependent
	Task Finishing (**TF**)	Yes	Dependent

Table 4-2 – Summary of the approaches

Method	Periodic entities	Required parameters	Calculated parameters	Specific restrictions
MD	Tasks	C, MP, MC	T, D, Ph	$\left\lceil \dfrac{D_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} + C_{Task} \leq T_{Msg}$
	Messages	C, T, D, PT, CTL	Ph	
MMF	Tasks	C, MP, MC	T, D, Ph	$\left\lceil \dfrac{MAXf_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} + C_{Task} \leq T_{Msg}$
	Messages	C, T, D, PT, CTL	Ph	
MF	Tasks	C, MP, MC	T, D, Ph	$\left\lceil \dfrac{f_{MsgP}}{t_{EC}} \right\rceil \times t_{EC} + C_{Task} \leq T_{Msg}$
	Messages	C, T, D, PT, CTL	Ph	
TD	Tasks	C, T, D, MP, MC	Ph	$\left\lceil \dfrac{D_{Task}}{t_{EC}} \right\rceil \times t_{EC} + C_{Msg} \leq T_{Task}$
	Messages	C, PT, CTL	T, D, Ph	
TMF	Tasks	C, T, D, MP, MC	Ph	$\left\lceil \dfrac{MAXf_{Task}}{t_{EC}} \right\rceil \times t_{EC} + C_{Msg} \leq T_{Task}$
	Messages	C, PT, CTL	T, D, Ph	
TF	Tasks	C, T, D, MP, MC	Ph	$\left\lceil \dfrac{f_{Task}}{t_{EC}} \right\rceil \times t_{EC} + C_{Msg} \leq T_{Task}$
	Messages	C, PT, CTL	T, D, Ph	

Table 4-3 – Summary of the parameters and restrictions of the approaches

The specific restriction for the method Message Deadline in Table 4-3 means that the deadline of messages is constrained to a value that is dependent upon the producer task execution time, C_{Task}, and is always inferior to the period. To allow the message deadline, D_{MsgP}, to be greater than that value, a buffering technique can be used as referred in Section 5.2. The same can be said about the specific restrictions of the other methods where the constraint can be either the Deadline, or the Maximum Finishing Time or the Finishing time.

The assessment factors for the choice of a parameter determination method can be:

- The most constrained resources, either the network or the nodes;
- Independent scheduling;
- The buffer memory usage.

4.3.5 Overlapping approach

When the execution window is dependent of the transmission window of a related message this means that the release instant of a task is dependent of the release instant of the related message.

Parameter restrictions

According to the type of interactive task, several restrictions apply in dependence of the messages produced and/or consumed, namely:
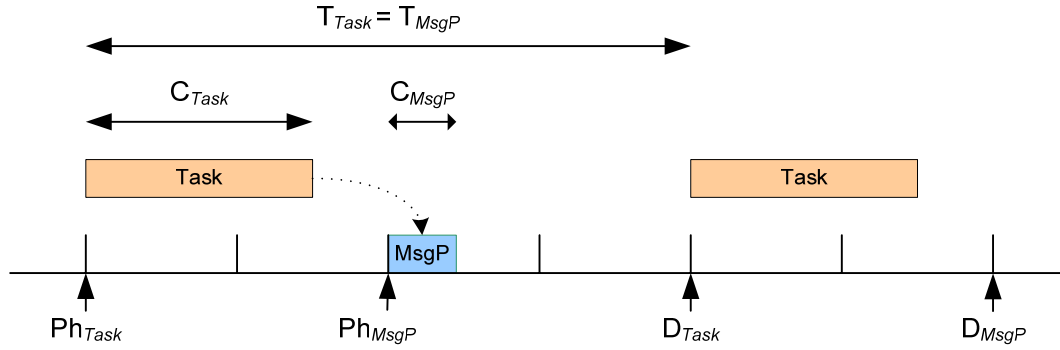
- **Producer tasks**

Figure 4-25 – O: Task produces a message

Considering a producer task *Task* that produces a message *MsgP* (see Figure 4-25), maximizing the execution window of *Task* and the transmission window of *MsgP*, the parameters of *Task* and *MsgP* are derived as follows:

$$T_{Task} = T_{MsgP} = T_{DS}$$

(4-74)

$$Ph_{MsgP} = Ph_{Task} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-75)

$$D_{Task} = D_{MsgP} = T_{DS}$$

(4-76)

- **Consumer tasks**
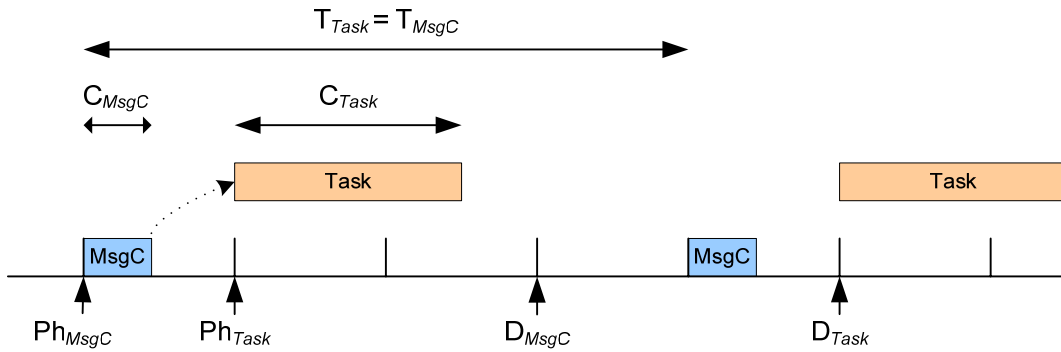
Figure 4-26 – O: Task consumes a message

Figure 4-26 shows a consumer task *Task* that consumes a message *MsgC*. The general restrictions for a consumer task are:

$$T_{Task} = T_{MsgC} = T_{DS}$$

(4-77)

$$Ph_{Task} = Ph_{MsgC} + \left\lceil \frac{C_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-78)

$$D_{MsgC} = T_{DS} - \left\lceil \frac{C_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-79)

$$D_{Task} = T_{DS}$$

(4-80)

- **Consumer/producer tasks**
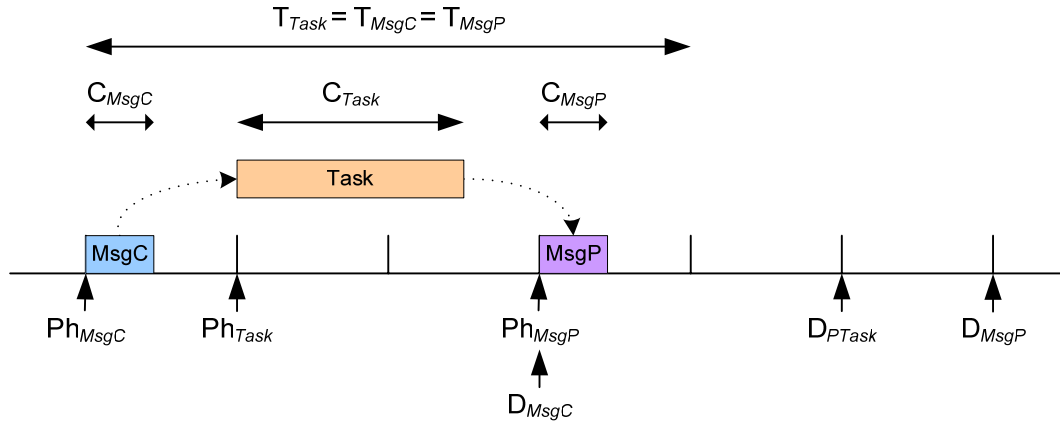
Figure 4-27 – O: Task that consumes and produces a message

Figure 4-27 shows a consumer/producer task *Task* that consumes a message *MsgC* and produces a message *MsgP*. The general restrictions for a consumer/producer task are:

$$T_{Task} = T_{MsgC} = T_{MsgP} = T_{DS}$$

(4-81)

$$Ph_{Task} = Ph_{MsgC} + \left\lceil \frac{C_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-82)

$$Ph_{MsgP} = Ph_{Task} + \left\lceil \frac{C_{Task}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-83)

$$D_{MsgC} = T_{DS} - \left\lceil \frac{C_{MsgC}}{t_{EC}} \right\rceil \times t_{EC}$$

(4-84)

$$D_{Task} = T_{MsgP} = T_{DS}$$

(4-85)

4.3.6 Comparing the various approaches

Due to the choice of having the release instant of tasks independent of the release instants of related messages, the following benefits were identified:

- The release jitter of tasks and messages is reduced;
- An online change of the parameters is less likely to force a data stream recalculation.

The lower levels of release jitter result directly from the choice of having independent windows of successive entities in a data stream. This means that, for a task, any jitter that may occur results only from higher priority tasks executing at the same node. For such a task, any release jitter that occurs on consumed messages has no influence on it. The same can be said about any message release jitter where the release jitter of producer tasks has no influence on it. The approaches cannot also give guarantees about jitter but they can reduce it in an automated way, which can be integrated into an admission control system. If the maximum jitter is known beforehand, the use of the proposed approaches can help in admitting system changes that do not make the jitter too high, while rejecting the others.

The Message, or Task, Maximum Finishing approach promises the best results in terms of the end-to-end delay. In online systems, this approach might require a recalculation every time a system change occurs. If this becomes unacceptable, then the Message, or Task, Deadline approach might be better. This approach has the worse end-to-end delay but, for many applications, this is not as important as the jitter. This approach gives better results in a changing environment because the system can accommodate extra load without a significant impact on the jitter. This is due to the extended transmission, or execution, windows. This is justified because this approach offers isolation between the execution and transmission windows of consecutive entities in a data stream.

The online change of parameters is less likely to force a data stream recalculation. An example can be a message that has its transmission changed to a different EC within its transmission window. If the execution window of the consumer task does not overlap the transmission window of this message then its parameters do not have to be calculated.

None of the approaches tries to distribute the bus, or node, load across the different ECs. For example, in case there is a message that has to be transmitted in an EC after its release EC, a lower jitter can be achieved by using some sort of procedure time-shifting. With

such a technique, load could be distributed so that jitter is minimized. Therefore, these approaches do not try to minimize jitter with any other technique not detailed in this chapter. Nevertheless, other techniques that might be complementary to the ones presented in this work can be very useful.

4.4 Real-time procedures

A procedure is a conceptual set of operations that might be implemented through a set of tasks, where each operation does not necessarily map to each task. This partitioning should take into consideration aspects like: separation of functionality, location of needed resources and load balancing between the nodes. Due to the partitioning, some tasks might be allocated to specific nodes in the case where specific resources are required, and these resources are only available at a certain node. Other tasks might be less stringent and their allocation is possible within a set of nodes.

In order to illustrate the concept of real-time procedures, two examples of the control of a wheelchair, the RobChair, are used [CSF05]. These are: the control loops for the speed control of each wheel and the collision detection.
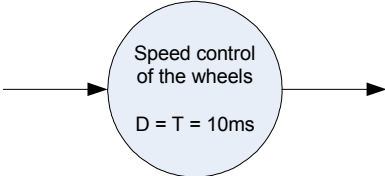
Figure 4-28 – Real-time procedure of the speed control of the wheels

The speed control of the wheels, shown in Figure 4-28, consists in a set of tasks, namely, encoder count acquisition, displacement calculation, position determination, new set point determination and actuation (see Figure 4-29).
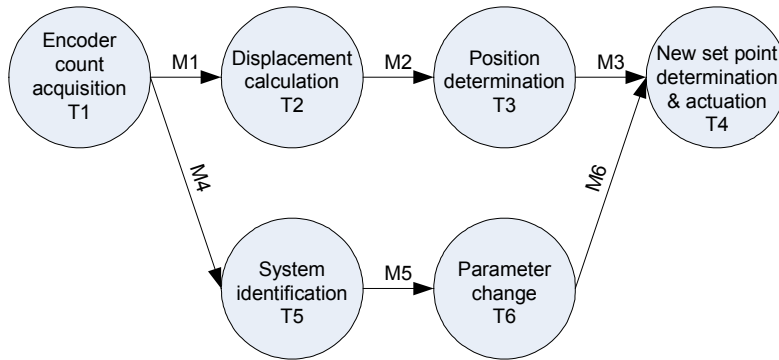
Figure 4-29 – Set of tasks for the speed control of the wheels

Other tasks can also be identified such as the system information and the parameter change. The sampling period, i.e., the reading of the encoders is of 10ms. A further requirement is the delay between sampling and actuation which must be either a small fraction of the sampling period or as close as possible of a full sampling period [Cer99].
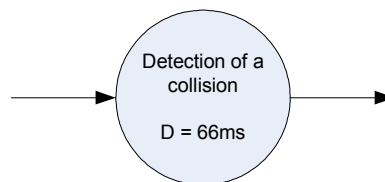
Figure 4-30 – Real-time procedure of the collision detection

The collision detection can also be divided into different tasks such as the event triggered detection at the sensors node, the data processing to decide if the event is significant and, if a decision to react is taken, the issuing of the commands to stop the motion. Here the real-time requirements are the maximum end-to-end delay. In this case it can be 66 ms.
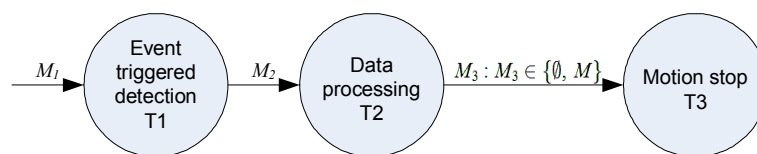
Figure 4-31 – Real-time procedure of the collision detection

These two examples show clearly that the timeliness requirements found in a real-time system may be defined at a higher level than the task level. The resulting data streams are still independent of the system partitioning but help in a first definition of constraints for tasks and messages.

A real-time procedure [CSF05] has, at least, one timing constraint. Usual constraints are the period and the deadline, but others can also be defined.

The period defines a common value for the period of each task and message of the procedure.

A deadline constraint limits the completion of the operations of a particular procedure. If a real-time procedure is mapped to a single task then the deadline of the procedure becomes the deadline of the task. If, on the other hand, a real-time procedure is mapped to various communicating tasks, then the issue is how to define the deadlines of the involved tasks and messages. If a procedure has a deadline parameter, this implies that the available time must be distributed between the deadline parameters of tasks and messages. The procedure's deadline acts as a maximum end-to-end delay for the correspondent set of tasks and messages.

Another constraint that is relevant for control procedures is the actuation instant. This constraint requires that the actuation is done as close as possible to one or several specific time instants. This is very relevant because a controller is designed for a particular actuation instant, relative to the acquisition. If the actual actuation is in the form of a message then, by using a buffer, the message can be stored until the appropriate instant comes and its transmission is issued. If, on the other hand, the actual actuation is in the form of a task, which means that it is a task that actually controls the process, then its release has to be postponed until an instant where there is still time to execute it. Logically, the execution window of this task should be as small as possible so that a low jitter is guaranteed. Therefore, an algorithm to map this constraint to the correspondent tasks and messages should begin by defining the initial phase of the actuation entity, either a task or a message, and its deadline.

After the definition of the system architecture and having the tasks allocated to the nodes, these constraints might be further refined as explained in the approaches presented in section 4.3.

For the case where a procedure is mapped to more than one task, an approach has to be chosen in order to define the timing parameters of each, correspondent, task and message. Simple algorithms will now be proposed in order to accomplish the mapping of parameters for the three main approaches previously presented in this chapter.

4.4.1 Mapping procedure parameters for net-centric and node-centric approaches

If a procedure has a period parameter, this implies that each correspondent task and message will inherit the same value for the period.

If a procedure has a deadline parameter, this implies that the available time must be distributed according to the net-centric approach. In summary, the net-centric approach tries to favour the transmission window of messages while constraining the execution windows of tasks.

A simple algorithm to distribute the available time resulting from a procedure's deadline begins with allocating the available time to each entity so that, in ideal conditions, they can be executed, or transmitted. The amount of time to distribute is equal to the execution, or transmission, time rounded up to the next EC. If, after this, there is any remaining time then it should be distributed among the messages until their limit is reached. A possible limit would be to have, as maximum, the deadline of each message equal to their period. If every message has its deadline at the maximum then any remaining time should be distributed among the tasks.

If a procedure has an actuation instant, then, after defining the initial phase and deadline of the actuation entity, as previously explained, its initial phase can be considered as a deadline for all other tasks and messages. This available time can be distributed as if it was a procedure deadline (previously described in this section).

The mapping of the procedure's parameters for a node-centric approach is very similar to the one presented for a net-centric approach. The difference is that the execution windows are favoured while the transmission windows are constrained. Therefore, the algorithms are similar apart from the exception noted.

4.4.2 Mapping procedure parameters for an overlapping approach

If a procedure has a period parameter, this implies that each correspondent task and message will inherit the same value for the period.

If a procedure has a deadline parameter, then the available time can be used to maximize the deadlines of the individual tasks and messages.

If a procedure has an actuation instant, then after defining the initial phase and deadline of the actuation entity, the available time can be used to maximize the deadlines of the other entities.

4.5 Application of the data stream analysis and the real-time procedures

The data stream analysis is beneficial in different phases of a distributed system planning. This analysis is useful in situations like: computational and network load evaluation, early stage system parameters definition, real-time procedures, when the various entities don't have specific deadlines; systems with a high network load, where the messages have strict deadlines; systems with a high node load, where the tasks have strict deadlines.

At an early stage where very few parameters are defined apart from the execution times and message size, this analysis helps in the definition of suitable values for the initial phases and the deadlines. After this, the system developer can fine tune any parameter according to some special needs. If, on the contrary, almost every parameter is clearly defined, then this analysis is useful to check the assumptions.

Also the data streams can be useful in the definition of the necessary message buffering. This topic is covered in the next chapter.

A real-time procedure is implemented by a set of tasks that communicate through message passing. Primary objectives regarding the timeliness of a procedure constitute the main concern. But, secondary objectives, like loosing the constraints of more constrained resources or extra gains from trying to finish the procedure earlier than necessary, should also be taken into account. For instance, these secondary objectives might be formulated towards a simpler admission of new tasks. The admission of new tasks might involve reviewing the data streams in order to accommodate the extra node and network load.

Real-time procedures allow closing the gap between the system design and the implementation. At the system design level, procedures and their parameters are specified. In dependence of the information gathered during the analysis phase, the procedures might be more, or less, complex in terms of the tasks involved. But when these procedures map to various tasks, the determination of the tasks' and messages' parameters is not trivial. This technique helps to reduce the time from design to implementation in real-time systems.

5 Scheduling and dispatching with FTT-CAN

This chapter presents implementation solutions for building and analyzing the data streams towards the holistic scheduling of tasks and messages. It is also shown how to determine the required level of message buffering and expand the FTT-CAN so that tasks can also be dispatched.

5.1 Building and analyzing the data streams

5.1.1 Building the data streams

A possible algorithm for data stream construction is presented in Figure 5-1. The first step is checking the parameters of the tasks and the messages. Then the data streams are created according to the structure depicted in Figure 5-1. The next step is the verification of the period and deadline of every entity in the various data streams. If there are no contradictory values, like having two entities in the same data stream with different periods, the remaining parameters are determined in the final step.

The main steps of this algorithm will be further discussed in the following sub-sections.
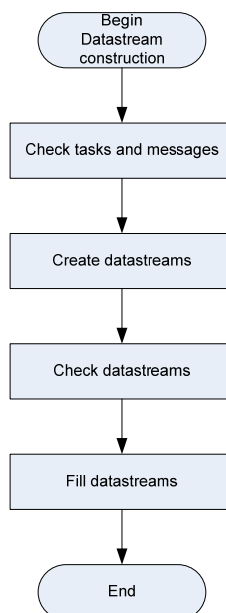
Figure 5-1 – Algorithm for data stream construction

A data stream represents a single stream of information that is extracted from the graphs that represent the interaction between tasks. A conceptual structure to store the data streams is shown in Figure 5-2.

This structure is mostly made of linked lists. The top linked list stores a data stream per position. Then, each node of a data stream stores a task, *PTask*, and a linked list of produced messages, *LProd*. *LProd* is a linked list of produced message sets, *AND Produced Sets*. Each of these sets can have alternative sets of produced messages, *OR Produced Sets*. In the end, the message sets that are produced are equal to the number of positions of the AND Produced Sets, because only one of each OR Produced Sets is actually transmitted.

This way, this data streams structure supports the scenarios previously presented.
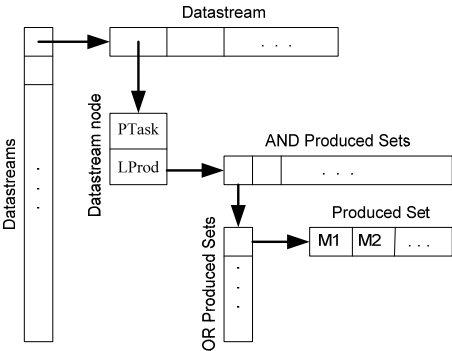
Figure 5-2 – Structure to store data streams

The extraction of the data streams is an iterative process. This process is depicted in Table 5-1. The main procedure, *FindDataStreams*, searches the task list looking for producer tasks and also looking for non-produced messages. These two types of entities mark the beginning of a new data stream. Then, for each data stream the iterative procedure, *FindPrecedences*, follows the chain of interaction between tasks, creating new data streams where appropriate.

```
FindDataStreams:
  FOREACH task FROM global task list
    IF task IS ProducerTask
      Create datastream
      Add datastream to global datastream list
      FindPrecedences( datastream, task )
  FOREACH task FROM global task list
    IF task IS ConsumerTask
      IF consumed message sets HAVE non-produced messages
        Create datastream
        Create datastream node
        Add datastream node To datastream
        Add datastream to global datastream list
        FindPrecedences( datastream, task )

FindPrecedences:
  IF task IS ProducerTask
    Create list of consumer tasks
    FOREACH ConsumerTask FROM current task
      IF iteration>1
        Create datastream from previous
      Create datastream node
      Add task To datastream node
      Add datastream node To datastream
      FOREACH produced message set
        Create prodset
        FOREACH message FROM current prodset
          IF message IS consumed by current ConsumerTask
            Add message to prodset
      FindPrecedences( Current datastream, ConsumerTask )
    FOREACH produced message set FROM current task
      IF produced message sets HAVE non-consumed messages
        Create datastream from previous
        Create datastream node
        Add datastream node To datastream
        Add datastream to global datastream list
  ELSE
    Create datastream node
    Add task To datastream node
    Add datastream node To datastream
```

Table 5-1 – Algorithm to create data streams

5.1.2 Interdependence analysis

As explained in the previous chapter, when a task participates in more than one data stream the calculated initial phase, *Ph*, for each data stream is, most probably, different. But in order to be able to check these *Ph* values, an interdependence analysis must be done. This analysis of the data streams must check the existence of closed loops.

The solution proposed is based on building an ordered task list. An algorithm to accomplish the interdependence analysis is presented in Figure 5-3. The most challenging step in the algorithm is checking if a task can be inserted in the ordered task list. This requires the analysis of the various data streams where the tasks, which are already in the list, participate. The algorithm finishes successfully when all tasks have been inserted in the list. If at least one task can not be inserted in the list then the algorithm aborts because a closed loop was found.
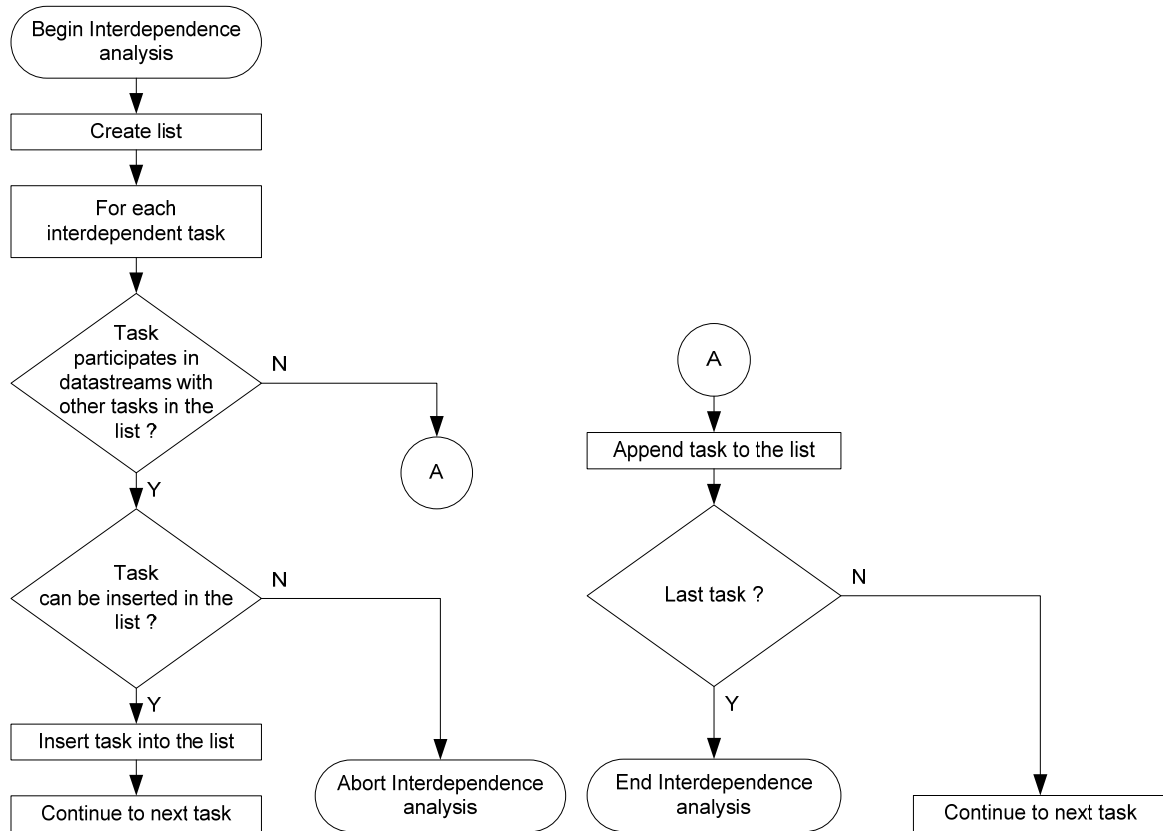
Figure 5-3 – Algorithm for checking the existence of closed loops

5.2 Message buffering

When a message is produced its transmission cannot be done immediately so that the global timing restrictions can be met according to a holistic schedule. According to the schedule, a message is transmitted in an EC after the production EC. This message, that has been produced but is waiting to be transmitted, is said to be in transit. A message in transit must be stored in a transmission buffer during the interval between its production instant and the instant where the last related consumer task starts running. An example of the usage of a transmission buffer can be seen in Figure 5-4, where a message P can be written to the transmission buffer after the deadline of its previous instance. Considering that the task's execution time is equal to its WCET, the task termination is always after the deadline of the previous instance of message P.
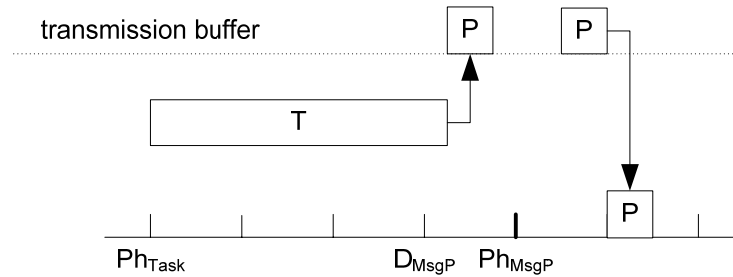
Figure 5-4 – Use of a transmission buffer

The number and size of messages in transit varies from EC to EC. The identification of the buffer memory requirements depends on the analysis of the messages in transit. This analysis can be accomplished with the support of the data streams. The process of finding the buffer memory requirements can be done, at first, for each task. Considering the various productions of each task in a data stream, its transmission buffer must accommodate the largest message that can be produced. Then considering the nodes to which the tasks are allocated, the message buffering requirements for each node can be evaluated either on a static allocation basis, i.e. constant for the whole macro-cycle, or on a dynamic allocation basis where the requirements are defined on an EC basis for a macro-cycle. The flowchart is represented in Figure 5-5.

Considering the unicast scenario with the transmission of a single message, if a producer task finishes earlier than its WCET the transmission buffer might still be full. This might happen due to a message, which was generated by the previous task instance, which is still waiting to be transmitted in accordance with the schedule. In this case, in order to guarantee that a message is transmitted in the expected EC, another buffer, message buffer, is required for every producer task.

A production buffer can be used to solve this issue. This buffer has to accommodate a full message.

The production buffering allows a certain decoupling between tasks and messages. An example of the usage of a transmission buffer and a production buffer can be seen in Figure 5-6, where a message P is produced before the task's WCET. In this situation the transmission buffer might still be storing the previous instance of message P, therefore, the new message P is stored in a production buffer and is later transferred to the transmission buffer.

The same rationale can be used in the case of a message that is consumed by more than one task. In this scenario, the message has to be stored in a buffer until the last consumer task reads it.

This analysis has to be accomplished for each node in the system so that in the end a map with the required buffering level for each EC is produced.
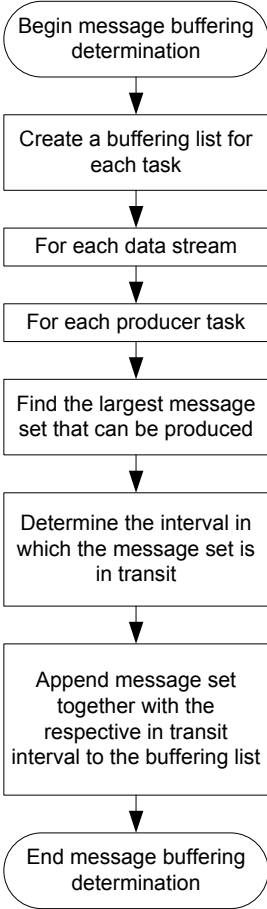
Figure 5-5 – Message buffering determination
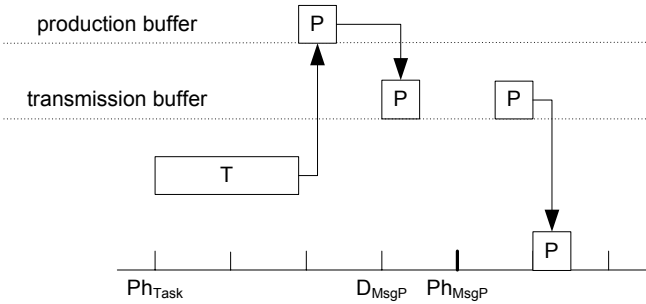
Figure 5-6 – Use of a transmission and production buffers

In summary, message buffering is beneficial in the following situations:

- To give an architectural support to a holistic schedule where messages may be scheduled for transmission in an EC different from the one when they have been produced;

- To consider the possibility of a task finishing on an EC before its WCET if the deadline of the previous produced message has not expired;

- To remove any deadline constraint imposed by a parameter determination method.

From an architectural point of view the message buffering is implemented by the kernel in a reserved memory space. After knowing the message buffering requirements for each EC of the macro-cycle, a design decision regarding the memory allocation has to be taken. On one hand static allocation can be used where the maximum required memory for each node is allocated during system initialization. And on the other hand dynamic allocation can be used, where the allocated memory at each node changes according to the requirements. The static allocation simplifies the memory management reducing the computation overhead while the dynamic allocation reduces the memory requirements. Overall, the determination of the level of buffering can be important for distributed systems that include nodes with low memory availability.

5.3 Task dispatching

5.3.1 Overview

The dispatching is the specific operation of instructing the resource to start processing an entity selected by the scheduling algorithms. A solution for the dispatching of real-time periodic tasks and messages can be based on a centralized dispatcher.

In order to not overload the existing nodes (Stations), another special node (Master) is added to the system to accomplish this goal (see Figure 5-7).
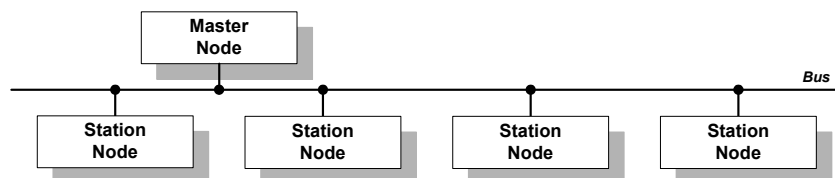
Figure 5-7 – Master/station nodes architecture

The Master triggers the execution of tasks in the stations and the exchange of messages in the bus, in a time-triggered manner.

Each Station node acts upon a trigger event and, in accordance, dispatches any task or message. The Stations have a variable number of tasks to be executed and can produce both synchronous and asynchronous/sporadic messages.

The bus acts as a triggering vehicle for tasks and messages.
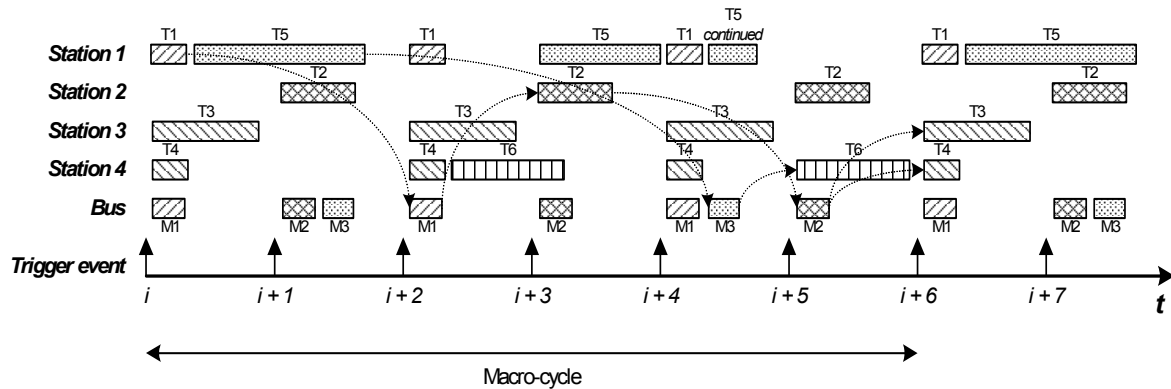
Figure 5-8 – Triggering of task execution and message sending

The example in Figure 5-8 shows a typical producer/consumer environment with preemption at the node level of task 5 by task 1 (with higher priority). In every station the task dispatching and the message production is synchronous with the trigger event. The data flow of this scenario is the one depicted in Figure 5-9.
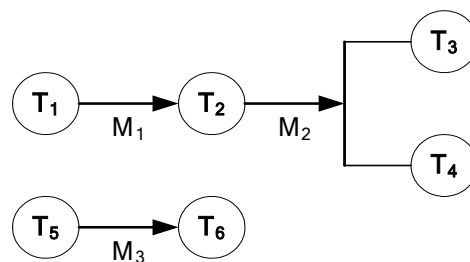
Figure 5-9 – Example data flow

5.3.2 Node architecture

In order to present the node architecture, the type and layers of the Master and Station kernels have to be described [CSFM06]. These have to be distinguished because they will have different functionality.

Kernel of the master

The kernel of the Master node will be responsible for the centralized dispatching of tasks and messages; therefore it will have a modified dispatcher. This dispatcher will interact not with the local tasks and messages but with the tasks and messages at the station nodes by sending a trigger message. This kernel can also be responsible for the holistic scheduling of tasks and messages at the Station nodes, but not necessarily. The kernel of a Station node will not have a scheduler or even dispatcher.

In general terms, some distinction is made between an operating system and a kernel. The former, often including a kernel, is usually more elaborate, possibly including drivers for specific hardware such as disc drives and networking. The latter, when a distinction is made, refer to the most basic part of the system that tracks priorities and manages task switching. A kernel that supports real-time tasks is called a real-time kernel. The development of real-time kernels for embedded systems is covered in [Sch99].

The previous description of a kernel can be further refined according to the available functionality. A characterization of different levels of basic functionality, in increasing order of complexity, can be resumed as follows [Lap97]:

- Nano-kernel – Simple thread-of-execution (same as "flow-of-control") management. It essentially provides only one of the three services provided by a kernel; that is, it provides for task dispatching.

- Micro-kernel – A nano-kernel that provides for task scheduling.

- Kernel – A micro-kernel that provides for intertask synchronization and communication via semaphores, mailboxes, and other methods.

Considering this characterization, the kernel of the Master can be considered a micro-kernel, if it includes the scheduler, or otherwise it can be considered a nano-kernel. This kernel has 4 layers as depicted in Figure 5-10.

Application Programming Interface		
Scheduling (Optional)	Dispatching (Trigger event generation)	
List management		
Context switch	Interrupt handling	Clock handling

Figure 5-10 – Layers of the kernel in the master node

Based on a schedule, the Master coordinates the exchange of messages on the bus and the task dispatching at each Station through a special trigger event. The dispatching unit is the responsible for the trigger event generation. This trigger event is in the form of the trigger message (TM). An optional scheduling unit may be functioning in the master node. If this kernel has no scheduling then the schedule must be supplied by another node. A specialized co-processor such as the one described in [MNF02] can be an alternative. The lower layers are responsible for the basic kernel support of the above operations.

Kernel of the station

According to the previous characterization, the kernel of each Station node is a nano-kernel, where the dispatcher receives the information about the tasks and messages to be dispatched from the trigger message. This kernel has 4 layers as depicted in Figure 5-11.

Application Programming Interface			
Dispatching	Termination	Sending	Receiving
Task Handling		Message Handling	
List management			
Context switch	Interrupt handling		Clock handling
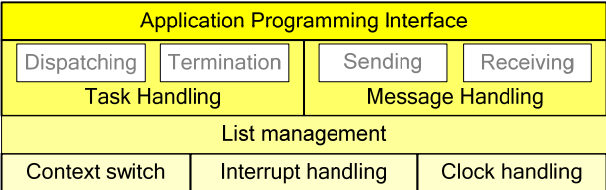
Figure 5-11 – Layers of the kernel in the station node

The task handling has 2 major units: the dispatcher and the termination. The dispatcher acts upon 2 types of events: the trigger event (generated through a trigger message, TM) from the master node and the task finish event. These events are signaled through an interrupt. So the dispatcher is interrupt-driven. The termination is called whenever a task finishes. If this function is called through an interrupt then a monitoring task can be used to take care of the execution time verification.

The message handling has 2 major units: the sending and the receiving. The sending unit acts upon a system call requesting the sending of a message. When the TM that instructs the transmission of this message arrives, the message is posted to the transmission buffer and this event is logged. When the message transmission finishes the actual transmission time is checked against the expected transmission time (ETT) by the monitoring. The ETT has been previously determined based on the message length and on the protocol particularities (e.g. fields, bit stuffing, …). The receiving unit acts upon an interrupt

signaling a message arrival. The monitoring can check that the message has arrived within an acceptable time window.

Upon a trigger event, the kernel dispatches any task or message that it is instructed to. The interval between trigger events becomes the time slice for this kernel.

From the trigger event, various scenarios can occur with task dispatching:

- No task is selected for execution;

- A task is selected for execution, and the Station was idle;

- A task is selected for execution, but there is another task still running;

- More than one task is selected for execution.

Due to the inexistence of a scheduler in the Stations, the kernel just has to act in accordance with the trigger event (see Figure 5-12) and execute, respectively, the following procedures:

- Keep doing the same, either idle or running a task;

- Start execution of the selected task;

- Preempt the running task, start execution of the selected task and, upon its termination, resume the previous task;

- Start the successive execution of each selected task.

Task preemption can occur as planned in the schedule. This possibility allows the system to achieve a high responsiveness permitting higher priority tasks to preempt others.
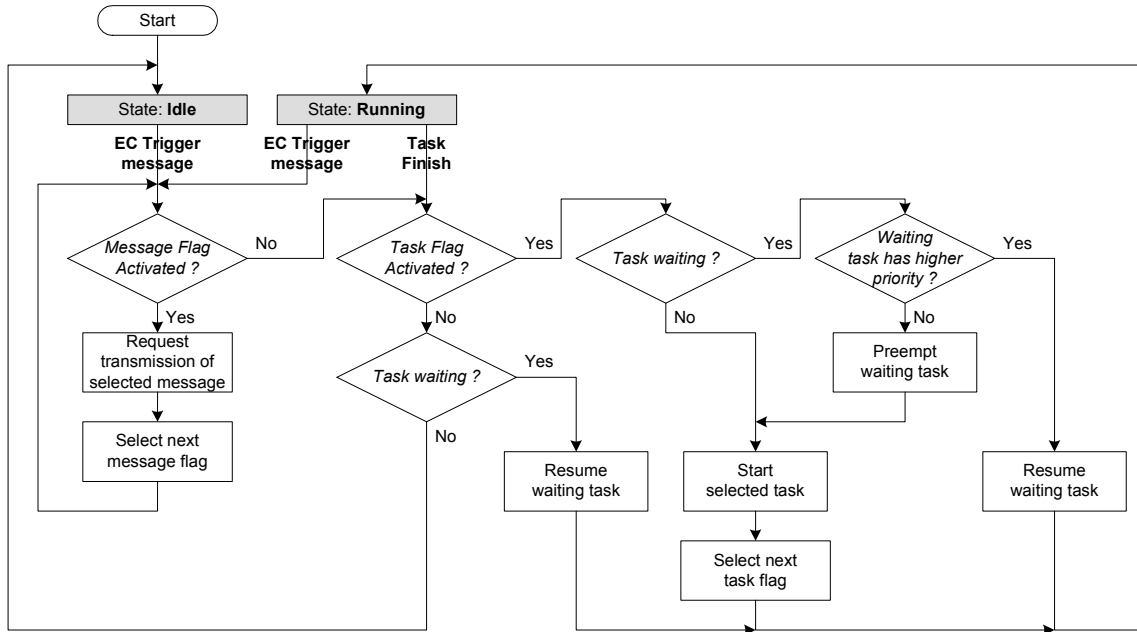
Figure 5-12 – Station kernel flowchart

The existence of more than one task selected for execution is a way to use a possible dead time between the end of the execution of the first task and the next trigger event. The order of execution of a set of selected tasks is pre-determined and can take into consideration characteristics like: priority, execution time, period, deadline, etc...

From the trigger event, any message selected for dispatching instructs the kernel to transmit that message.

The granularity of the trigger event should be tuned with system parameters like: task and message periods, average size of tasks, average size of messages, number of tasks and messages, etc...

Tasks and messages can be loaded through the common bus, maintaining the connectivity needs to a minimum.

A general view of the kernels across the nodes is shown in Figure 5-13.

Master node

Application Programming Interface		
Scheduling (Optional)	Dispatching (Trigger event generation)	
List management		
Context switch	Interrupt handling	Clock handling

Network . . .

Station node 1

Application Programming Interface			
Dispatching	Termination	Sending	Receiving
Task Handling		Message Handling	
List management			
Context switch	Interrupt handling	Clock handling	

Station node *n*

Application Programming Interface			
Dispatching	Termination	Sending	Receiving
Task Handling		Message Handling	
List management			
Context switch	Interrupt handling	Clock handling	

Figure 5-13 – System view depicting the kernels

Currently this architecture can be implemented using the Controller Area Network [Can96] and the FTT-CAN protocol [APF02] to trigger the dispatching. The FTT-CAN protocol is presented in the next chapter. It is also possible to use FTT-Ethernet [PAG02] for the same purpose.

5.3.3 Using FTT-CAN for task dispatching

The FTT-CAN protocol relies in the system operator to correctly characterize the messages. Parameters like the message period are dependent not only on the physical process, but also on the worst case execution time of both the producer task, and the consumer task.

Each task has its own characteristics, namely: execution time, running node and type of interaction with other tasks. So, in order to achieve a feasible schedule for the messages, tasks have also to be considered in a holistic scheduling.

In a system with several tasks running in several nodes, a node may have more than one task assigned. The problem is how to schedule tasks in order to accomplish the intended global schedulability without increasing the system complexity, i.e. maintaining a low overhead.

To accomplish the goal the EC trigger message must be redesigned to accommodate also in its data field, apart from the synchronous messages that must be transmitted and additional

coding, an indication of which tasks must be started in the current EC. This way, the EC trigger message can be used to also trigger remote task execution. The trigger message data field has now to accommodate two flag areas, one for tasks and another for messages. Each bit from the task data field is a flag that indicates whether a task must be dispatched in the current EC, or not. In a similar way, each bit from the message data field is a flag that indicates whether a message must be transmitted in the current EC, or not.
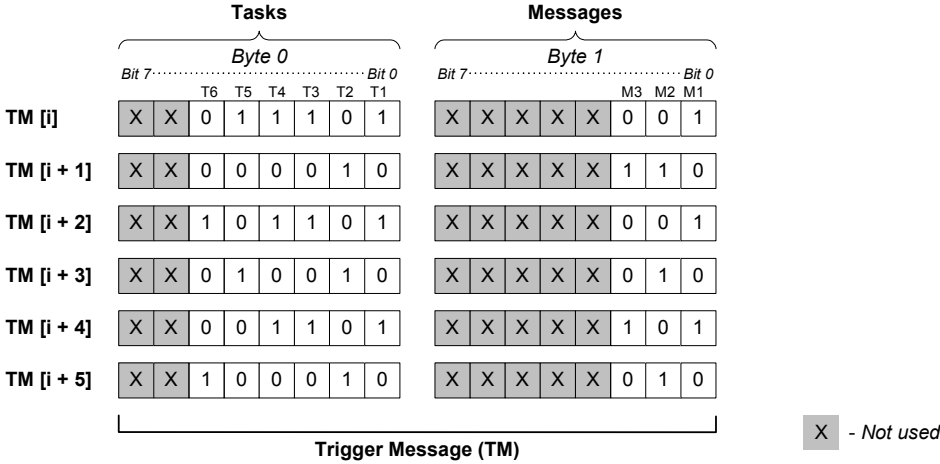
Figure 5-14 – EC Trigger Message data contents

Recalling the example from Figure 5-8, the trigger message could have a data field with 2 bytes, one byte for tasks and another one for messages, as shown in Figure 5-14.

The definition of the data field can be done during the initialization phase but if the sum of the number of tasks of synchronous messages and the length in bits of the codes exceeds 63 (limit of CAN2.0A) then an extended trigger message must be used. For small systems one trigger message should be enough, but for larger systems various extended trigger messages could be used.

The discretization of time imposed by the EC trigger message puts a constraint over the task and message parameters:

- Tasks – T_i and Ph_i have to be rounded to an EC multiple (note that C_i can be less than the EC duration);

- Messages – T_m and Ph_m have to be rounded to an EC multiple (this restrictions were already presented and thoroughly discussed in FTT-CAN based systems).

This extension to the FTT protocol for the dispatching of tasks in the nodes was proposed in [CF02]. With it, the Master node assumes control of the overall distributed system,

triggering tasks in nodes and messages in the bus using the trigger message referred above. Some issues on Task Dispatching and Master Replication in FTT-CAN where further explored in [FFC$^+$02].

Using the FTT-based approach, the Master node can control the execution of tasks and the transmission of the messages associated with the data streams from producer to consumer tasks. This is done by defining a scheduling timeline, coordinating the dispatching of the producer tasks, the corresponding messages and the consumer tasks. This way, the FTT-based approach is able to guarantee the real-time behaviour of the system. It should just be referred that this timeline can be changed dynamically using the adequate mechanisms to avoid jeopardizing the real-time performance.

This technique was successfully used in the CAMBADA robot [SMA$^+$05].

6 SimHol: a simulation and configuration tool

The simulation of distributed real-time systems plays an important role both for the offline scheduling and for testing online scheduling with scenario changes. A simulator must offer a suitable test bed for this kind of systems providing an accurate, as possible, view of the real working system. Various approaches to parameter determination, as well as different algorithms and architectures should be supported.

The main goals of a simulator can be:

- Support for the different entities of a distributed real-time system like tasks, messages, nodes and networks;

- Ability to adapt to different requirements, namely the scheduling algorithms and the architectural constraints;

- Support to tasks and messages that are not fully specified by offering various approaches for their determination;

- Output of significant data that helps to shorten the development cycle;

- Input and output of data at different stages of the simulation allowing a seamless integration with other tools.

In order to achieve these goals the simulator has to be developed in a modular way so that meaningful data is always accessible to other tools. Naturally, this openness will come at the cost of some performance penalty, because data will have to be translated to an open and non-volatile form such as a plain text file.

The purpose of the simulator, named SimHol [CF03] that stands for Holistic Simulator, is to support the joint scheduling of messages and tasks in distributed embedded systems. This simulator tries to match the goals previously described and to offer a simple graphical interface. The approaches taken in the development of this simulator were already presented by the authors in a previous work [CF02]. The SimHol is also useful in supporting the system partitioning, in terms of which tasks are allocated to which nodes, and the tuning of tasks and messages' parameters so that a feasible schedule can be achieved.

The development of the SimHol was based on a traditional approach where the development process can be described with three stages:

- Analysis: defining the scope of the problem to be solved
- Design: creating an overall structure for a system
- Implementation: writing and testing the code

This process has an iterative nature and this was also the case with the development of the SimHol.

These three stages will be explained after a brief overview of other simulators.

For the purpose of this work, special attention has also been paid to the study of holistic simulators, either coming from the academic world or from the industrial world.

Henderson, *et al*. [HKRB98] have developed a design tool, Xrma, that supports the schedulability analysis of hard, uni-processor and distributed real-time systems. Xrma automates the analysis and supports the performance verification of diverse real-time systems composed of tasks executing on multiple processors which communicate using the CAN fieldbus. Vector Informatik GmbH has developed CANoe/DENoe [Vec02]. This is a tool that supports the entire development process for networked systems from planning to implementation. Due to its open architecture, CANoe/DENoe is able to solve complex tasks and is tailored for special applications. Models both graphic and text-based as well as evaluation windows are provided for simulating and analyzing entire distributed networks. This tool supports protocols like CAN, TTCAN and FlexRay.

A simulator, called TrueTime, that facilitates the simulation of the temporal behaviour of a multitasking real-time kernel executing controller tasks was presented by Henriksson, *et al*., [HCA03]. True time is a MATLAB toolbox and is more focused on dynamic real-time control systems. While real-time tasks can be scheduled according to a selected algorithm [AHC05], the scheduling of messages is not considered in the simulation. Therefore a holistic scheduling is not possible.

Until now, as far as the authors are aware, a centralized holistic scheduling and dispatching tool has not been presented. Therefore, this work presents a solution in the form of a simulator, named SimHol, that derives several task parameters and also allows the use of different approaches to the task and message scheduling. The SimHol uses the flexible time-triggered (FTT) paradigm for scheduling both tasks and messages, and supports

different network architectures and scheduling algorithms. This approach is different from all the others, as far as we know.

6.1 Analysis

During the analysis stage two sets of requirements were identified, namely: internal and external.

The internal requirements were:

- The use of the FTT model as presented previously;
- Support for different networks, like the Controller Area Network (CAN);
- The network connects various fixed nodes;
- Various tasks are executed concurrently at system level and at node level;
- Tasks can be stand-alone or interactive;
- Interactive tasks communicate using the message passing paradigm;
- The interactions between tasks are represented as precedence graphs, data streams;
- Tasks and messages are defined according to the equations defined in Chapter 2;
- Messages may impose restrictions to the tasks, and vice-versa;
- Tasks and messages are dispatched according to the FTT paradigm;
- Support for different scheduling algorithms like the Rate Monotonic (RM), the Deadline Monotonic (DM), the Earliest Deadline First (EDF) and the Least Laxity (LL);
- Separate scheduling algorithm for tasks and messages, meaning that tasks can be scheduled with an algorithm that is different from the one used in message scheduling;
- Tasks and messages are scheduled using a centralized approach;
- The scheduler can consider system tasks that introduce overhead, like the kernel execution and the context switch.

The external requirements were:

- A simple interface with a visual tree-like representation of the data streams and a visual representation of the scheduling map;
- Input scenarios using plain readable text files;
- Output of all simulation data to plain readable text files;

- The possibility of editing the tasks' and messages' parameters;

- Fast response and execution time.

The core of the SimHol is constituted by two units: parameter determination and scheduling. The block diagram is depicted in Figure 6-1.
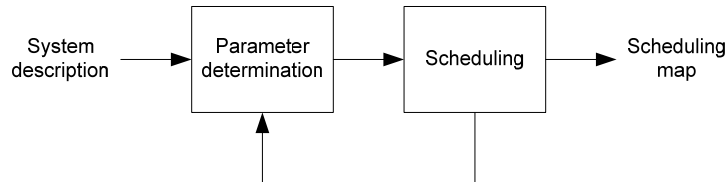
Figure 6-1 – Block diagram of the SimHol

The purpose of the parameter determination unit, depicted in Figure 6-2, is to determine any remaining unspecified parameters.
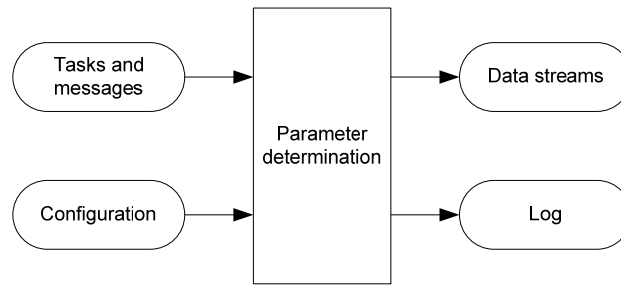
Figure 6-2 – Block diagram of the parameter determination unit

This unit takes as inputs the tasks, messages and configuration parameters. The configuration parameters can be used to guide the determination of any remaining unspecified parameters. The results of the parameter determination unit are the data streams.

The purpose of the scheduling unit is to attempt the scheduling of the tasks and messages. The block diagram is depicted in Figure 6-3.
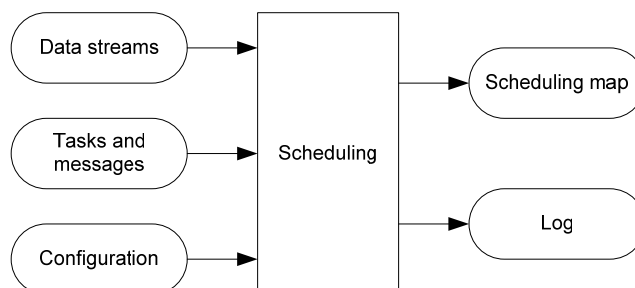
Figure 6-3 – Block diagram of the scheduling unit

In case of a successful scheduling a scheduling map is produced. In case of failure in the scheduling, resulting from a missed deadline, the information is feedback to the parameter determination unit so that a new scenario can be built. This unit takes as inputs the data streams, the list of tasks and messages and the configuration parameters. The data streams were generated by the parameter determination unit. The configuration parameters include the algorithms for the task and message scheduling.

6.2 Design

When designing the SimHol one of the main goals was design for change. That is we were looking for:

- Flexibility
- Extensibility
- Portability

In order to reach this goal the main design choice was the selection of an object-oriented approach due to its well known advantages. Therefore, the design steps can be resumed as:

- Find classes
- Specify operations
- Specify dependencies
- Specify interfaces

In terms of interoperability with other tools we were also looking for an easy integration with them. These tools could include a system layout designer, a data stream designer or a scheduling analyser.

The design is now presented from the point of view of the static diagrams and the flowcharts of the two main units: the parameter determination and the scheduler.

6.2.1 Object-oriented approach

The simulator was developed using an object oriented approach. This approach brings great benefits to the software development, maintenance and future upgradeability.

The static diagram depicts various classes arranged in three groups:

- Simulator and Scheduler
- Resources: Node and Bus
- Entities: Task and Message

The static diagram showing simplified classes (stripped of more internal attributes and functions) is depicted in Figure 6-4.

The simulator class, *CSimulator*, stores the major data structures like the lists of entities, data streams and scheduling map. This class is the main gateway to access all the functionality so it includes all the top level functions that are called from the Graphical User Interface (GUI). The GUI and I/O functions, like reading from files, are in separate classes isolating the core classes from the Operating System dependent functionality. With this design choice the simulator can be easily adapted to other execution environments. The scheduler is explained in the next section.
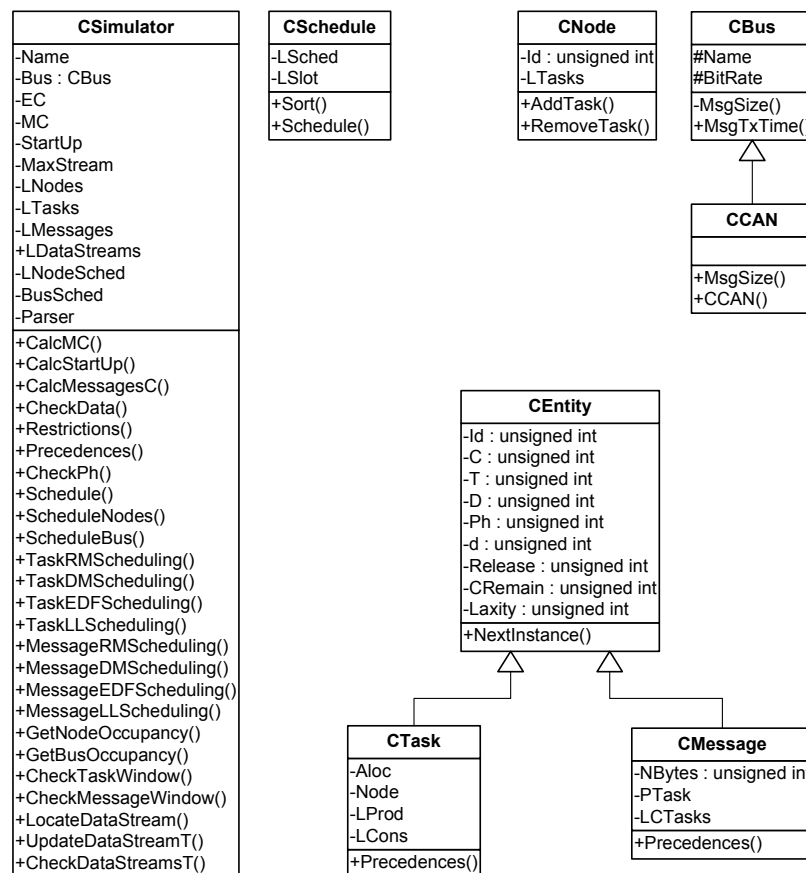
Figure 6-4 – Static diagram of the simulator

The resources are disputed by the correspondent scheduling units. Each node represents a computational node of the system (or a station node). For each node, allocated tasks are scheduled; and for the bus, messages are scheduled. The nodes are represented by the class *CNode* that basically manages the list of tasks allocated to it. The bus is represented by the

class *CBus*. In dependence of the type of bus, another class inherits from *CBus* and rewrites the method MsgSize(). In the current implementation of the SimHol, the CAN bus is already supported via the class *CCAN*.

The entities refer to the scheduling units, namely: tasks and messages. These entities are scheduled according to the resource they are using; nodes for the tasks and bus to the messages. Each entity, class *CEntity*, has two types of parameters. The general parameters, from *Id* to *Ph*, define the entity and are constant for every instance. The instance parameters, from *d* to *Laxity*, define each instance during the scheduling. The classes inherited from *CEntity* are *CTask*, representing a task, and *CMessage*, representing a message. The class *CTask* has mainly the lists of consumed and produced messages. The class *CMessage* has mainly the producer task and the consumer task list.

6.2.2 Parameter determination

The parameter determination unit is responsible for the determination of some of the tasks, or messages, parameters, namely: the period (T), the relative deadline (D) and the initial phase (Ph). This operation depends upon the approach selected. In a net-centric approach, messages impose constraints upon the tasks, while on a node-centric approach the opposite happens. The operation of the parameter determination unit for a net-centric approach is depicted in Figure 6-5.
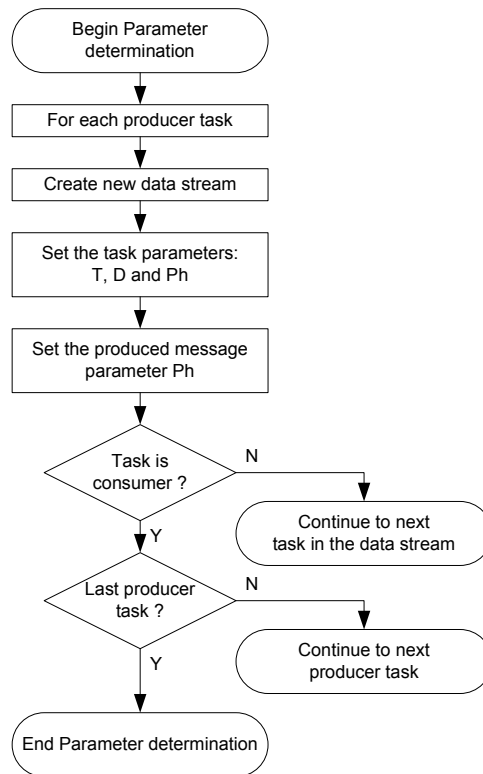
Figure 6-5 – Flowchart of the parameter determination unit for a net-centric approach

6.2.3 Scheduler

The scheduler is responsible for scheduling tasks at each node and messages at the bus. This goal is accomplished on an EC basis, which means that for each EC of the scheduling interval the scheduler considers the scheduling units, tasks and messages, of each resource, nodes and bus. The scheduling interval is given by the sum of both start-up and macro-cycle intervals. The macro-cycle is defined as the smallest interval where a scheduling pattern is found. When no changes occur in the system, this pattern repeats indefinitely with the exception of an initial period, called the start-up, where there are still tasks, or messages, that have not been scheduled [CF02].

The scheduler has the following main features:

- Selectable static or dynamic priorities;
- Selectable sorting criteria with ascending or descending order;
- Selectable scheduling interval;
- The possibility to break the task execution, or message transmission, for several ECs, that may not be adjacent;

- The scheduler parameters are defined independently for the task set and for the message set;
- The scheduling interval is organized as a list of free slots, that represent the available time intervals, and a list of scheduled entities, that represent time intervals already taken by some task or message.

The sorting criteria can be selected from the task, or message, general parameters or from the instance parameters, i.e. the parameters from a particular instance of a task, or message. The list of possible general parameters is: execution/transmission time (C), period (T), relative deadline (D) and initial phase (Ph). And the list of possible instance parameters is: absolute deadline (d), release instant, remaining execution/transmission time and laxity. Also the sorting order can be ascending or descending. With all these possible combinations the more popular algorithms can be easily used. The Table 6-1 presents some examples regarding the algorithms: rate monotonic, deadline monotonic, earliest deadline first and least laxity.

	Rate Monotonic RM	Deadline Monotonic DM	Earliest Deadline First EDF	Least Laxity LL
Priorities	Static	Static	Dynamic	Dynamic
Criteria	Period	Relative Deadline	Absolute Deadline	Laxity
Order	Ascending	Ascending	Ascending	Ascending

Table 6-1 – Scheduling parameters for some algorithms

The scheduler can be invoked upon any resource, node or bus. So, for a full system scheduling the scheduler is invoked once for each node and another time for the bus. This shows that this scheduler is suitable for other system architectures that make use of various buses. From the point of view of the scheduling all that is necessary is to schedule each new resource.

On creation several parameters must be specified:

- Priority, that can be static or dynamic
- Preemption (yes or no)
- Sorting criteria and direction (ascending or descending)
- List of entities to be scheduled

- Duration of the simulation (number of ECs)
- EC duration
- Allow multiple entities per EC (yes or no)
- Entities may be broken (yes or no, this is more suited to the messages)

Tasks can be executed across several ECs. Because of this, a task may be preempted due to the arrival of another task with higher priority. If desired, a message can be transmitted across several ECs. In this situation the kernel has to be aware of this possibility and break the message in smaller pieces (a sort of transport layer must be provided). This overhead has to be considered. This technique may be acceptable when the majority and most frequently used messages are short and a few, but there can be much longer messages even if rarely used. In this case the EC might be chosen to best suit the needs of the short messages but without this possibility the transmission of long messages might become impossible.

These parameters allow a great flexibility for the scheduling procedure and provide headroom for further functionality.

During the initialisation phase, two lists are created with a position for each EC:
- Free slots list (on each EC)
- Scheduled entities list (on each EC)

The free slots list has an entry for each EC. Each entry is a list of the free slots on the current EC. The parameters of each slot are the EC relative beginning and ending instants. This type of data structure allows the existence of scattered free time slots on the EC. The scheduled entities list has also an entry for each EC. Each entry is a list of the entities that were scheduled in the current EC. The parameters of each scheduled entity are the identification of the entity, and the EC relative beginning and ending instants. With these two lists it is possible to make the scheduling close to reality because other time constraints imposed by the system can be taken into account. For instance, the kernel execution time at the beginning of the EC and context switches can be considered by the scheduler.

The constructor of the scheduler is shown on Figure 6-6.
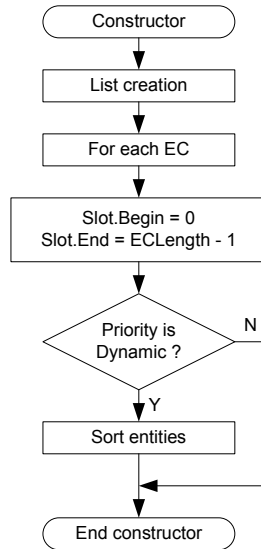
Figure 6-6 – Flowchart of the scheduler constructor

Apart from initializing the list, the type of priority is checked. If a static priority is chosen then entity sorting will take place at this phase. Otherwise, if a dynamic priority is chosen then entities are sorted every EC.

The scheduling operation may be invoked any time after the scheduler object has been created. This means that the scheduling operation may be executed several times and for different time intervals. Nowadays it is more common to execute the scheduling operation for the whole simulation. The scheduling operation is depicted in Figure 6-7.

For each EC, the priority type is checked. And again, if the priority is dynamic then the entity list for this resource is sorted according to the criteria specified on creation time. After this, for each entity, the deadline is checked and in case of a missed deadline the scheduler reports the situation and stops. If the deadline is met, then, if the entity is scheduled on the current EC and there is still a free slot that is large enough, the entity is added to the scheduled entities list and the slot is updated or removed.

Scheduling continues until either the simulation ends or a deadline is missed. If the deadline was reached and the task is unable to finish in the current EC then the scheduler stops due to a missed deadline.
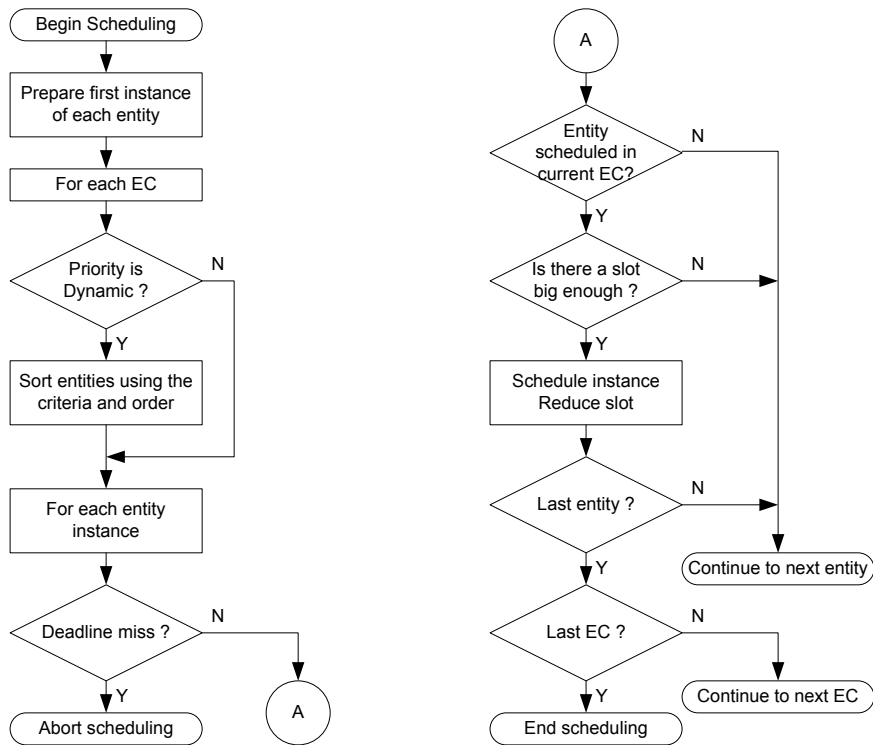
Figure 6-7 – Flowchart of the scheduling operation

6.2.4 Simulation scenario

A scenario is a set of parameters that define both the simulation environment and the simulation entities. The input data for a simulation scenario is defined by a Document Type Definition (DTD). A DTD must include information about the node and bus architecture, simulation options, tasks and messages. The DTD for a scenario is depicted in Table 6-2.

```
<!ELEMENT simulation
(name,bus,elementary_cycle,scheduling_algorithm,nodes,tasks,messages)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT bus (name,bitrate)>
<!ELEMENT bitrate (#PCDATA)>
<!ELEMENT elementary_cycle (#PCDATA)>
<!ELEMENT scheduling_algorithm (scheduling_tasks,scheduling_messages)>
<!ELEMENT scheduling_tasks (#PCDATA)>
<!ELEMENT scheduling_messages (#PCDATA)>
<!ELEMENT nodes (node+)>
<!ELEMENT node (name)>
<!ATTLIST node
          id ID #REQUIRED>
<!ELEMENT tasks (task+)>
<!ELEMENT task (c,t,d,ph_req,allocation_type,node_id)>
<!ATTLIST task
          id ID #REQUIRED>
<!ELEMENT c (#PCDATA)>
<!ELEMENT t (#PCDATA)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT ph_req (#PCDATA)>
<!ELEMENT allocation_type (#PCDATA)>
<!ELEMENT node_id (#PCDATA)>
<!ELEMENT messages (message*)>
<!ELEMENT message (data_bytes,t,d,ph_req,prod_task_id,cons_tasks)>
<!ATTLIST message
          id ID #REQUIRED>
<!ELEMENT data_bytes (#PCDATA)>
<!ELEMENT prod_task (#PCDATA)>
<!ELEMENT cons_tasks (cons_task+)>
<!ELEMENT cons_task (#PCDATA)>
<!ATTLIST cons_task
          id ID #REQUIRED>
```

Table 6-2 – Document Type Definitions (DTD) for a scenario

6.3 Implementation

In terms of implementation, the main goals were:

- The use of the object-oriented C++ programming language;

- The use of plain text files, such as XML, to store the scenarios and the scheduling map, instead of other proprietary formats;

- Function operation as close as possible to an actual working system;

- Clear separation between the core elements and the user interface, allowing an easy porting to other platforms;

- Exception handling, providing a good fault coverage resulting in a smooth operation.

Due to the speed advantage, the simulator was developed in C++ programming language [Str97], as a win32 native application, by using the C++Builder [Bor02] Integrated Development Environment (IDE). This approach assures a swift implementation of all requirements and the resulting application has a fast response and execution time. All the features of exception handling were used to cover the critical areas of the code.

The user input to the simulation is grouped in a scenario file. A scenario is composed of two types of data, namely: system characterization and entity declaration. The system characterization has the following data: the simulation name, the bus type and bit-rate, and the elementary cycle length. The entity declaration has the following data: the scheduling algorithm for tasks and for messages, the nodes, the tasks and the messages. The scenario file can be in a proprietary plain text format or in XML (see corresponding DTD in Table 6-2). The XML approach leads to a standardized input file format defined by a Document Type Definitions (DTD) file and also makes the development of a parser easier [YBP+04] [YCB+04].

This version of the simulator only implements the net-centric approaches, message deadline and the message maximum finishing, according to Chapter 1.

6.3.1 Network

Currently, the SimHol has a built-in support for CAN (Controller Area Network). In order for this architecture to be integrated in the simulator some aspects have to be defined like: the procedure to calculate the actual message length given a certain number of data bits, and the procedure to calculate the message transmission time given a certain bitrate. The number of data bits is defined for each message and the bitrate is defined for each scenario. The formula used to determine the maximum message size in CAN was already presented in Section 3.3. This formula is built-in into the method MsgSize() of class CCAN. This method returns the number of bits actually sent, *MessageBits*, for a given data message, *DataBits*.

The message transmission time is determined with the method MsgTxTime(). This method makes use of the method MsgSize() and the chosen bitrate. The formula to calculate the message transmission time is trivial.

In the same way other architectures can be integrated.

112

6.3.2 Interface

The user is presented with an intuitive interface. This interface can be divided in the command interface and in the window interface. The command interface offers the various commands through a toolbar (except editing the tasks' and messages' parameters). The window interface is divided in two main areas: the left area is used for outputting operation results and the right area displays all information about the simulation.

All input and output of data are made through plain text files. A simulation scenario is loaded into the simulator through an XML file that follows the previous DTD (see section 6.2.4). Simulation data can be output to plain text files. This data includes precise information about:

- Computed tasks' and messages' parameters,
- Task execution and message transmission,
- Nodes and bus occupancy,
- Execution windows.

An example of a simulation data file is shown in Figure 6-8. In this example, each two lines have information about an EC, tasks are grouped according to the node to which they are allocated, messages are grouped in the end, together with the resource that they utilize and temporal information is relative to the EC. So for instance at EC 0, task 1 begins at offset 0 and ends at offset 520, and in the same EC task 5 begins at offset 520 and ends at offset 1000 leading to a full occupancy of the node (100%).

```
--- Scheduling Map ---
Entity    T      T      N    T      N    T      N    T      T      N     M      M      M     Bus
  Id      1      5      1    2      2    3      3    4      6      4     1      2      3
 Unit micros micros    % micros    % micros    % micros micros    % micros micros micros    %
------ ------ ------ ------ ------ ------ ------ ------ ------ ------ ------ ------ ------ ------
  EC
  0      0    520    100         0           0              0                                 0
       520   1000
  1           0     100          0           0              0                                 0
            1000
  2           0      70          0           0              0      0                          6
             704                                                  65
  3                  0           0           0              0                                 0

  4      0            52         0           0              0                                 0
       520
  5           0     100    0    80           0              0                   0             6
            1000        800                                                    65
  6           0     100          0           0              0      0                          6
            1000                                                  65
  7           0      18          0           0       0    100            0                    7
             184                                  1000                  75
  8      0            52         0           0       0     40                                 0
       520                                         404
```

Figure 6-8 – Extract of a scheduling map file

The tasks' and messages' parameters can be edited in the simulator. This makes possible an iterative try and test approach to the system scheduling within the simulator environment. A new scenario file with the new parameters can be output.

113

6.4 Upgrading

Currently all upgrades, to architectures or to algorithms, are accomplished through a source file that is added to the SimHol project and to minor changes in the interface. So this means that the project needs to be recompiled. In the future, the SimHol can be transformed in a distributed object system allowing the inclusion of new services without the need of further recompilation. The new architectures and algorithms could be loaded using a XML dialect.

6.4.1 Upgrading architectures

In order to add an architecture, a new class has to be defined. This class inherits the CBus class and has to implement the method *MsgSize(unsigned NBytes)* that returns the number of bits necessary to transmit a message in this architecture. The *Bus* class is as follows:

```
class CBus
{
  virtual unsigned MsgSize(unsigned NBytes);

protected:
  AnsiString Name;
  unsigned BitRate;

public:
  unsigned GetBitRate() { return BitRate; }
  AnsiString GetName()  { return Name; }

  unsigned MsgTxTime(unsigned);
  void CheckBitRate();
};
```

The CAN architecture class is defined as:

```
class CCAN : public CBus
{
  unsigned MsgSize(unsigned);
public:
  CCAN(unsigned BR) { Name="CAN"; BitRate=BR;}
};
```

6.4.2 Upgrading scheduling algorithms

In order to make available other scheduling algorithms, the following simple steps have to be taken:

If the criterion is not available already, then it has to be added to the attributes of *CEntity*;

A new function that creates a *CScheduler* object, with the new criteria, and invokes the scheduler has to be written;

The SimHol has to be recompiled.

6.4.3 Connecting with other software

Due to the plain text and normalized data output to files, it is easy to develop tools that read this data and make further analysis, allowing a simple integration on a broader software suite. The Figure 6-9 shows an extract of an output file containing simulation data.

```
--- Nodes List ---
N Id: 1
N Id: 2
N Id: 3
N Id: 4

--- Tasks List ---
T Id: 1 C:    520(  0) T:  4000(  4) D:  2000(  2) PhReq:    0(  0) PhMin:     0(  0) MaxExEnd:    520(  0) N: 1 Prod: 1
T Id: 2 C:    800(  0) T:  4000(  4) D:  2000(  2) PhReq:    0(  0) PhMin:  5000(  5) MaxExEnd:    800(  5) N: 2 Cons: 1 Prod: 2
T Id: 3 C:   1248(  1) T:  4000(  4) D:  1748(  1) PhReq:    0(  0) PhMin: 10000( 10) MaxExEnd:   1248( 11) N: 3 Cons: 2
T Id: 4 C:    390(  0) T:  4000(  4) D:   890(  0) PhReq:    0(  0) PhMin: 10000( 10) MaxExEnd:    390( 10) N: 4 Cons: 2
T Id: 5 C:   2184(  2) T:  5000(  5) D:  5000(  5) PhReq:    0(  0) PhMin:     0(  0) MaxExEnd:   2704(  2) N: 1 Prod: 3
T Id: 6 C:   1404(  1) T:  5000(  5) D:  3904(  3) PhReq:    0(  0) PhMin:  7000(  7) MaxExEnd:   1794( 18) N: 4 Cons: 3

--- Messages List ---
M Id: 1 C:     65(  0) T:  4000(  4) D:  3000(  3) PhReq:    0(  0) PhMin:  2000(  2) MaxTrEnd:     65(  2) PTask: 1 CTask: 2
M Id: 2 C:     75(  0) T:  4000(  4) D:  3500(  3) PhReq:    0(  0) PhMin:  7000(  7) MaxTrEnd:     75(  7) PTask: 2 CTask: 3: 4
M Id: 3 C:     65(  0) T:  5000(  5) D:  2500(  2) PhReq:    0(  0) PhMin:  5000(  5) MaxTrEnd:    140( 15) PTask: 5 CTask: 6
```

```
--- Execution Window Map ---
 EC    T     T     N     T     N     T     N     T     T     N     M     M     M
  0    1     5           0           0           0     0           0     0     0
  1  255     5           0           0           0     0           0     0     0
  2    0     5           0           0           0     0           1     0     0
  3    0     5           0           0           0     0           1     0     0
  4    1   255           0           0           0     0         255     0     0
  5  255     5           2           0           0     0           0     0     3
  6    0     5         255           0           0     0           1     0   255
  7    0     5           0           0           0     6           1     2     0
  8    1     5           0           0           0     6         255     2     0
  9  255   255           2           0           0   255           0   255     0
 10    0     5         255         255           0     0           1     0     3
 11    0     5           0           0           0     0           1     2   255
 12    1     5           0           0           0     6         255     2     0
 13  255     5           2           0           0     6           0   255     0
 14    0   255         255         255           0   255           1     0     0
 15    0     5           0           0           0     0           1     2     3
 16    1     5           0           0           0     0         255     2   255
 17  255     5           2           0           0     6           0   255     0
 18    0     5         255         255           0     6           1     0     0
 19    0   255           0           0           0   255           1     2     0
 20    1     5           2           0           0     0         255     2     3
 21  255     5           2           0           0     0           0   255   255
 22    0     5         255         255           0     6           1     0     0
 23    0     5           0           0           0     6           1     2     0
 24    1   255           0           0           0   255         255     2     0
 25  255     5           2           0           0     0           0   255     3
 26    0     5         255         255           0     0           1     0   255
 27    0     5           0           0           0     6           1     2     0
 28    1     5           0           0           0     6         255     2     0
 29  255   255           2           0           0   255           0   255     0
```

Figure 6-9 – Extracts of an output file

The output file is organized in two main areas: entities and maps. The first includes the characteristics of all nodes, tasks and messages in the system. The second includes the scheduling map and the execution window map. These maps are organized in the following way. The first column shows the current EC. The node columns (N) relate the sum of the tasks execution time in the current EC to the EC time. The last column (Bus) shows the bus utilization that relates the sum of the messages transmission time in the current EC to the EC time. The task columns (T) are organized according to the node where they have been allocated and are displayed to the left of their node.

In the next section a simple example is used to demonstrate the SimHol operation.

6.5 Experiments

In this section a scenario is used as input to the graphical simulator, SIMHOL, presented in [CF03a]. This scenario consists of 6 tasks, running in 4 nodes, which communicate using 3 messages. The data streams for this task set are represented in Figure 6-10.
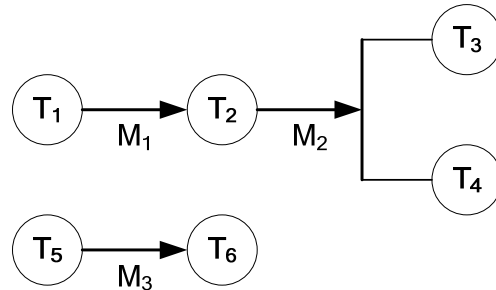
Figure 6-10 – Example data streams

This scenario is defined with the tasks' and messages' initial parameters.

```
--- Nodes List ---
N Id: 1
N Id: 2
N Id: 3
N Id: 4

--- Tasks List ---
T Id: 1        C:  520    Pri : 1    Node: 1    MsgProd: 1
T Id: 2        C:  800    Pri : 1    Node: 2    MsgCons: 1        MsgProd: 2
T Id: 3        C: 1248    Pri : 1    Node: 3    MsgCons: 2
T Id: 4        C:  390    Pri : 1    Node: 4    MsgCons: 2
T Id: 5        C: 2184    Pri : 1    Node: 1    MsgProd: 3
T Id: 6        C: 1404    Pri : 1    Node: 4    MsgCons: 3

--- Messages List ---
M Id: 1        C: 65      T: 4000    D: 3000    Pri : 1    PTask: 1    CTask: 2
M Id: 2        C: 75      T: 4000    D: 3500    Pri : 1    PTask: 2    CTask: 3: 4
M Id: 3        C: 65      T: 5000    D: 2500    Pri : 1    PTask: 5    CTask: 6
```

Table 6-3 – Example of a scenario

In Table 6-3, various tasks' and messages' parameters are shown. Each task is characterized with an execution time, priority, node of allocation and message consumed and/or produced. Each message is characterized with a transmission time, period, deadline, priority, producer task and consumer task(s).

The remaining parameters are determined by the analysis of the data streams and their values are depicted in the following table.

```
--- Tasks List ---
T Id: 1          T:  4000   D: 1000   Ph:     0
T Id: 2          T:  4000   D: 1000   Ph:  4000
T Id: 3          T:  4000   D: 1248   Ph:  9000
T Id: 4          T:  4000   D:  390   Ph:  9000
T Id: 5          T:  5000   D: 4000   Ph:     0
T Id: 6          T:  5000   D: 3404   Ph:  7000


--- Messages List ---
M Id: 1          Ph:  1000
M Id: 2          Ph:  5000
M Id: 3          Ph:  4000
```

Table 6-4 – Parameters derived due to the restrictions imposed by the messages

In Table 6-4, various derived tasks' and messages' parameters are shown. In bold are the deadline values that were obtained for each task.

This task and message set is schedulable using both the message deadline approach and the message maximum finishing approach.
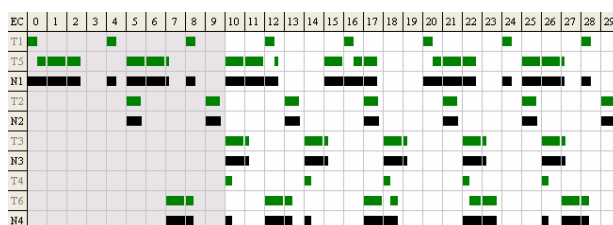
Figure 6-11 – Schedule using the message deadline approach

The schedule using the message deadline approach is depicted in Figure 6-11. This figure is a graphical representation of the execution time of each task and the load at each node for every EC of the simulation. The simulation lasts 30 ECs, where the startup phase is from EC 0 to EC 9 and the macro-cycle is from EC 10 to EC 29. The tasks are grouped above their node of allocation and are ordered according to their priority (given, in this example, by the Earliest Deadline First algorithm), with the highest priority task above. The load at each node is also represented, and is given by the sum of the execution times of the various tasks that are scheduled on each EC.

After scheduling using the message deadline approach the tasks' Maximum Execution End (MaxExEnd) and messages' Maximum Transmission End (MaxTrEnd) parameters are calculated. These values are shown in Table 6-5.

```
--- Tasks List ---
T Id: 1          MaxExEnd:   520
T Id: 2          MaxExEnd:   800
T Id: 3          MaxExEnd:  1248
T Id: 4          MaxExEnd:   390
T Id: 5          MaxExEnd:  2704
T Id: 6          MaxExEnd:  1794

--- Messages List ---
M Id: 1          MaxTrEnd:    65
M Id: 2          MaxTrEnd:   140
M Id: 3          MaxTrEnd:   205
```

Table 6-5 – Parameters calculated after scheduling using the message deadline approach

Now, using these values, MaxExEnd and MaxTrEnd, the constraints of the task set can be relaxed. The new tasks' deadlines and initial phases resulted from the maximum finishing approach are shown in Table 6-6.

```
--- Tasks List ---
T Id: 1        D:  4000   Ph:     0
T Id: 2        D:  3800   Ph:  5000
T Id: 3        D:  4248   Ph: 10000
T Id: 4        D:  3390   Ph: 10000
T Id: 5        D:  6000   Ph:     0
T Id: 6        D:  5404   Ph:  7000
```

Table 6-6 – Parameters derived after the message maximum finishing approach

It can be seen that the new tasks' deadlines are significantly higher.

The schedule generated by the message maximum finishing approach is depicted in Figure 6-12.
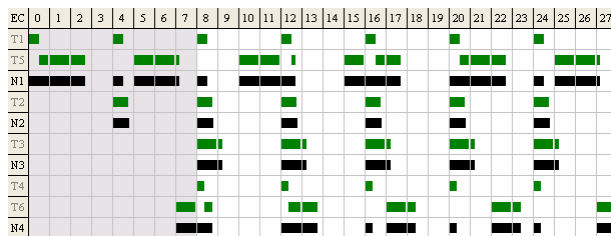
Figure 6-12 – Schedule using the message maximum transmission approach

These results clearly show the improved execution windows for tasks in the case of the maximum transmission approach. Starting with a pessimistic set of tasks' deadlines, this approach led to a much more relaxed set of deadline parameters. These improved execution windows can be used to facilitate the scheduling on nodes that also have stand alone tasks allocate.

6.6 SimHol reviewed

After developing the first version of the SimHol and conducting some experiments, a new approach to the simulation of distributed real-time systems has arisen. The main goals, previously presented were extended to the following:

- Support for the different entities of a distributed real-time system like tasks, messages, nodes and networks;
- Ability to adapt to different requirements, namely the scheduling algorithms and the architectural constraints;
- Support to tasks and messages that are not fully specified by offering various approaches for their parameter determination;
- Support for the data streams including the interdependence analysis;
- Allow online changes such as changes to the parameters and to the set of entities;
- Support for real-time procedures and specifically to control procedures;
- Output of significant data that helps to shorten the development cycle;
- Standard input and output of data at different stages of the simulation allowing a seamless integration with other tools.

While the first version had an integrated visual interface, this second version does not rely on this type of interface but one can easily be integrated as an independent tool. All meaningful data is output using an XML format. This is the main reason why the integration with other tools is simple.

The development of this version of the SimHol was based on a traditional approach where the development process can be described with three stages:

- Analysis: defining the scope of the problem to be solved
- Design: creating an overall structure for a system
- Implementation: writing and testing the code

This process has an iterative nature and this was also the case with the development of this version of the SimHol.

These three stages will now be explained.

6.6.1 Analysis

The internal requirements defined for the previous version have been augmented with the following new requirements:

- Support for the concept of a special task, called procedure, that unfolds to a data stream;

- Data streams can now be multi-dimensional where a node of a data stream can represent another data stream;

- Take into account the interdependencies between data streams during parameter determination;

- Inclusion of other parameter determination techniques that allow less specified tasks and messages;

- Each module of the project must have its own stand-alone test.

On the other hand, the external requirements defined for the previous version are replaced by the following:

- The simulator should be a set of independent batch tools that communicate through files;

- All files used for input and for output are in standard XML and defined by a proper DTD;

- Fast response and execution time.

The core of the SimHol is now constituted by four units, namely: procedure expansion, parameter determination, task allocation and scheduling. The block diagram is shown in Figure 6-13.
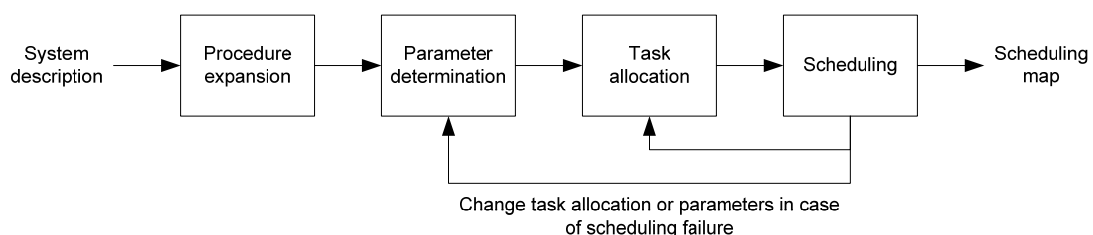
Figure 6-13 – Block diagram of the new SimHol

These four units will now be presented in the remaining of the section.

Figure 6-14 depicts the procedure expansion unit.
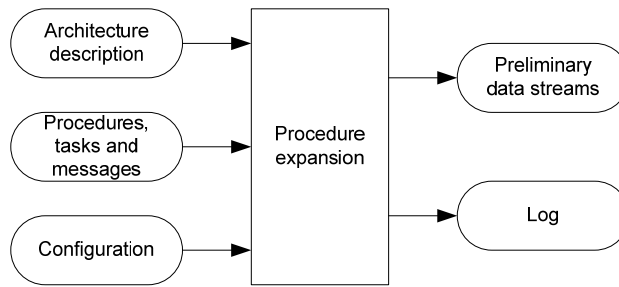
Figure 6-14 – Block diagram of the procedure expansion unit

This unit produces the preliminary data streams, with provisory parameters, after expanding the procedures. These data streams may not be fully specified. This unit takes as inputs: the architecture description; the list of procedures, tasks and messages; and the configuration parameters. The architecture description has information about the nodes and the interconnecting network, namely, the data rate and how to calculate the message length. The list of procedures, tasks and messages have all the available information about these entities. Some parameters like the period, or the deadline, might be undefined at this point. The configuration includes simulation parameters like the length of the elementary cycle (EC). The results of this unit are the preliminary data streams that include the involved tasks and messages.

Figure 6-15 shows the parameter determination unit.
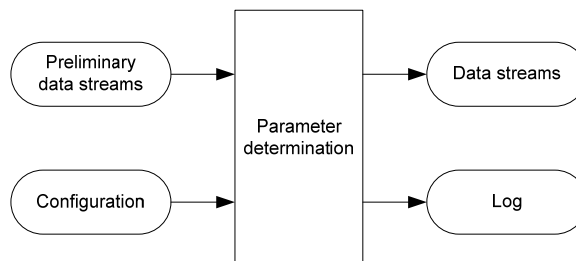
Figure 6-15 – Block diagram of the parameter determination unit

This unit as two purposes, the first is to determine any remaining unspecified parameters, and the second is to generate the final parameters after considering the interactions between data streams. This unit takes as inputs the preliminary data streams, which include the tasks and messages, and the configuration parameters. The preliminary data streams were generated by the system preparation unit. The configuration parameters can be used to guide the determination of any remaining unspecified parameters. The results of this unit are the fully specified data streams.

Figure 6-16 depicts the task allocation unit.

121
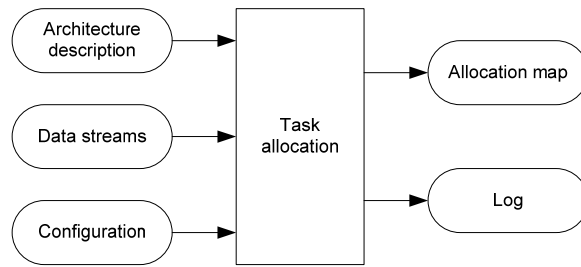
Figure 6-16 – Block diagram of the task allocation unit

The purpose of this unit is to try the allocation of tasks that were not assigned to a particular node. Using the information in the data streams, now fully specified, this unit can attempt a load sharing policy. This unit takes as inputs the architecture description, the fully specified data streams, which include the tasks and messages, and the configuration parameters. The configuration parameters are used to select and tune the policy for the task allocation. The results of this unit are the allocation map for the tasks.

Figure 6-17 shows the scheduling unit.
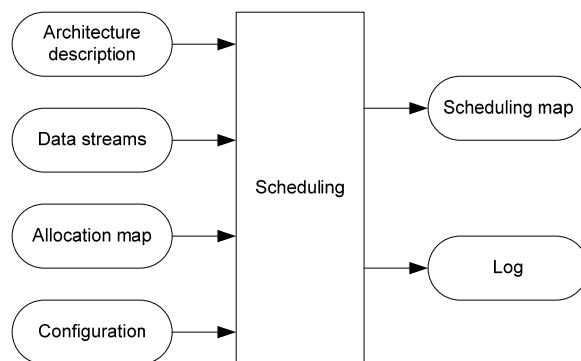
Figure 6-17 – Block diagram of the scheduling unit

The purpose of this unit is to attempt the scheduling of the tasks and messages. In case of a successful scheduling a scheduling map is produced. In case of failure in the scheduling, resulting from a missed deadline, the information is feedback to the task allocation unit and the parameter determination so that a new scenario can be built. This unit takes as inputs the architecture description, the fully specified data streams, which include the tasks and messages, the allocation map and the configuration parameters. The architecture description describes the resources and can contain information about fixed system software. The fully specified data streams were generated by the parameter determination unit and the allocation map was generated by the task allocation unit. The configuration parameters include the algorithms for the task and message scheduling.

6.6.2 Design

The design of this new version of the SimHol has basically the same goals.

The static diagram depicts various classes arranged in four groups:

- Simulator: ProcedureExpansion, ParameterDetermination, TaskAllocation and Scheduler
- Resources: Node and Bus
- Entities: Task and Message (and respective interface Entity)
- Data streams: DataStream, DataStreamNode, MessageSet and DSDependenceNode.

The static diagrams showing simplified classes (stripped of getters and setters, and of more internal attributes and functions) are depicted in Figure 6-18, Figure 6-19, Figure 6-20 and Figure 6-21.
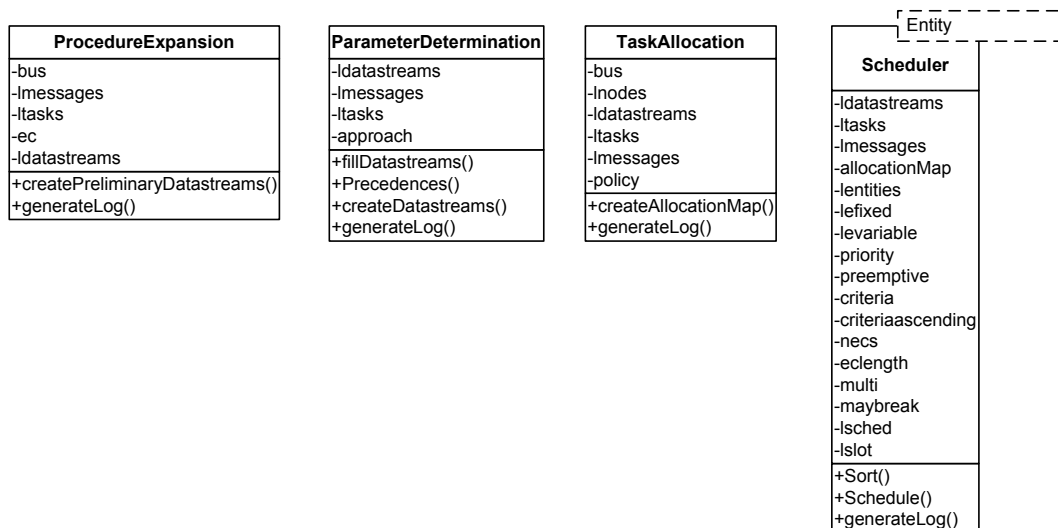
Figure 6-18 – Static diagram of the simulator main classes

The static diagram in Figure 6-18 shows each class that represents one of the units of the SimHol. An instance of each of these four classes is a stand-alone tool that reads inputs and writes outputs only to files. Another class that integrates these four can be developed so that the process becomes fully automated.

The class Scheduler is a template class, or generic class, which means that it has a parameter that is a type. This versatility is useful because it allows the same functionality to be used both for tasks and messages, in which case the type is, respectively, Task and Message. The functionality of the scheduler is the same of the first version of the SimHol.

123

Node
-id
-ltasks
+addTask()
+removeTask()

Bus
#name
#bitRate
+msgSize()

CAN
+msgSize()
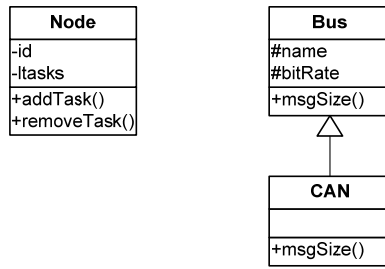
Figure 6-19 – Static diagram of the resource classes

Figure 6-19 shows the classes that represent the resources, namely the nodes and the bus. The class CAN is already available but others can easily be integrated.

«interface» Entity
+*getId()*
+*getC()*
+*setC()*
+*getT()*
+*setT()*
+*getD()*
+*setD()*
+*getPh()*
+*setPh()*
+*getd()*
+*setd()*
+*getRelease()*
+*setRelease()*
+*getCRemain()*
+*setCRemain()*
+*getLaxity()*
+*setLaxity()*
+*getPhReq()*
+*setPhReq()*
+*getMinSlot()*
+*setMinSlot()*
+*getMaxEnd()*
+*setMaxEnd()*
+*getMaxEndEC()*
+*setMaxEndEC()*
+*NextInstance()*

«interface» Entity

Message
-Id
-C
-T
-D
-Ph
-d
-Release
-CRemain
-Laxity
-TReq
-DReq
-PhReq
-MinSlot
-MaxEnd
-MaxEndEC
-nBytes
-PTask
-LCTasks
+isStandAlone()
+NextInstance()
+CheckForCTask()

«interface» Entity

Task
-Id
-C
-T
-D
-Ph
-d
-Release
-CRemain
-Laxity
-TReq
-DReq
-PhReq
-MinSlot
-MaxEnd
-MaxEndEC
-alloc
-node
-LProd
-LCons
+isStandAlone()
+NextInstance()
+getLProdLargestD()
+getLConsLargestD()
+setLProdPh()
+setLProdD()
+findUniqueCTasks()

Figure 6-20 – Static diagram of the entity classes

The classes shown in Figure 6-20 represent the entities, tasks and messages. Both entities have common parameters that will be referenced by the scheduler as the criteria. These parameters are accessed through a common set of getters and setters that are enforced by the interface Entity. This interface must be implemented by the classes Task and Message.

124

DataStream
-Idsnodes
-approach
-T
-MaxEnd
-MaxEndEC
+validateAndSetT()
+validateAndSetD()
+calculateTaskConstraintsD()
+calculateMessageConstraintsD()
+regionFlood()

DataStreamNode
-task
-Ph
-D
-LProd
-LProdPh
-LProdD
+getLProdLargestD()

MessageSet
-Imessage
+getLargestD()
+setPhAll()
+setDAll()

DSDependenceNode
-task
-Ids
-Idsnodes
+getLowestTaskPh()
+getLowestDSShift()
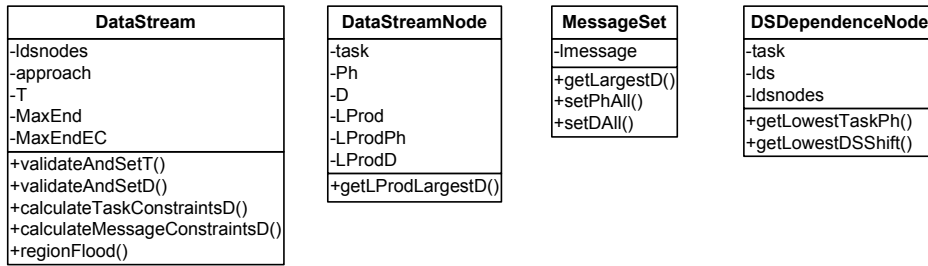
Figure 6-21 – Static diagram of the data stream classes

The last set of classes that is shown in Figure 6-21 is related to the data streams. For each data stream that is found an instance of the class DataStream is created. So, each instance stores the list of nodes of the data stream, the approach selected and some general parameters like the period of each entity. Each node is basically an instance of the class DataStreamNode. Each node can have, at most, a task and a list of produced message sets. Each position of this list refers to another list of alternative message sets. As previously explained, from these sets only one message set is actually transmitted. Each message set is basically an instance of the class MessageSet and includes a list of messages.

If a task participates in more than one data stream its parameters will depend upon the involved data streams. When this happens, an instance of the class DSDependenceNode is created. Each instance refers to a particular task and includes the list of data streams, where it participates, and the list of data stream nodes, where the task appears in each data stream.

6.6.2.1 Document Type Definitions for a scenario

Due to the modular approach used in the design of this version of the SimHol, the input and output data is now organized separately. The data is grouped as follows:

- Architecture description;
- The entities: Procedures, tasks and messages;
- Data streams;
- Configuration data;
- Simulation scenario.

All this data is organized according to a Document Type Definition (DTD) that will now be presented for each of the previous groups.

The architecture description refers to both hardware and software architectural aspects of the underlying system. The hardware aspects define parameters from the bus and the nodes, while the software aspects can define timing parameters from the running system

software at each node. Considering that the nodes can have different performances and capabilities, each node has separate timing definitions for running system software like the kernel. For example, the kernel is involved in at least two different instants, its execution is started at the beginning of the EC for fetching new dispatch instructions and internal management and regularly, according to the dispatching, to accomplish a context switch after the execution of a task. The EC relative system software has two parameters, begin and end, that define a fixed time interval of execution, while the task relative system software has one parameter that defines a fixed time interval immediately after the task stops. The format of the architecture description is organized as a DTD and is shown in Table 6-7.

```
<!ELEMENT architecture (bus,nodes)>
<!ELEMENT bus (name,bitrate)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT bitrate (#PCDATA)>
<!ELEMENT nodes (node+)>
<!ELEMENT node (name,ec_fixed,task_fixed)>
<!ATTLIST node
          id ID #REQUIRED>
<!ELEMENT ec_fixed (frames)>
<!ELEMENT frames (frame*)>
<!ELEMENT frame (begin,end)>
<!ATTLIST frame
          id ID #REQUIRED>
<!ELEMENT begin (#PCDATA)>
<!ELEMENT end (#PCDATA)>
<!ELEMENT task_fixed (#PCDATA)>
```

Table 6-7 – Architecture description DTD

The entities, tasks and messages, have several parameters. These parameters are the requested parameters and are subject to reevaluation by the parameter determination unit. The procedures can be specified using a description where they are defined like tasks. The only difference between a procedure and a task is that a procedure has a defining data stream with potentially numerous tasks and messages. The DTD that represents the entities is presented in Table 6-8.

```
<!ELEMENT entities (tasks,messages)>

<!ELEMENT tasks (task+)>
<!ELEMENT task
(name,c,t,d,ph,allocation_type,node_id,messages_consumed,messages_produced)>
<!ATTLIST task
            id ID #REQUIRED>
<!ELEMENT c (#PCDATA)>
<!ELEMENT t (#PCDATA)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT ph (#PCDATA)>
<!ELEMENT allocation_type (#PCDATA)>
<!ELEMENT node_id (#PCDATA)>
<!ELEMENT messages_consumed (and*)>
<!ELEMENT messages_produced (and*)>
<!ELEMENT and (or+)>
<!ELEMENT or (message_id)>
<!ELEMENT message_id (#PCDATA)>
<!ATTLIST or
            id ID #REQUIRED>

<!ELEMENT messages (message*)>
<!ELEMENT message (name,data_bytes,t,d,ph)>
<!ATTLIST message
            id ID #REQUIRED>
<!ELEMENT data_bytes (#PCDATA)>
```

Table 6-8 – Entity description DTD

When a task refers to a procedure, this procedure can be further refined using the same description in separate documents, one for each procedure, where the name of the document comes directly from the procedure name. The identifiers from the procedures, tasks and messages have a global scope within the simulation scenario.

The description of the data streams has two parts: the involved entities and the actual data streams. The description of the involved entities is in accordance with the DTD previously presented (see Table 6-8), while the actual data streams are output according to the format defined in the DTD presented in Table 6-9. The description of the involved entities includes now the calculated parameters of the tasks and messages. This way the requested parameters continue to exist in a separate document. The tasks and messages with the calculated parameters can easily be imported into a new document so that they can be used in a future simulation.

127

```
<!ELEMENT datastreams (datastream+)>
<!ELEMENT datastream (datastream_node+)>
<!ATTLIST datastream
        id ID #REQUIRED>
<!ELEMENT datastream_node (task_id,d,ph,messages_produced,m_prod_d,m_prod_ph)>
<!ATTLIST datastream_node
        id ID #REQUIRED>
<!ELEMENT task_id (#PCDATA)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT ph (#PCDATA)>
<!ELEMENT messages_produced (and*)>
<!ELEMENT and (or+)>
<!ELEMENT or (message_id)>
<!ELEMENT message_id (#PCDATA)>
<!ATTLIST or
        id ID #REQUIRED>
<!ELEMENT m_prod_d (#PCDATA)>
<!ELEMENT m_prod_ph (#PCDATA)>
```

Table 6-9 – Data stream description DTD

This information is conveyed from the internal structure as depicted in Figure 5-2. It can be seen that the DTD representation closely resembles the internal structure.

The configuration data is used by all units of the simulator and is described by the DTD shown in Table 6-10.

```
<!ELEMENT configuration (elementary_cycle,approach,allocation_policy,scheduling_algorithm)>
<!ELEMENT elementary_cycle (#PCDATA)>
<!ELEMENT approach (#PCDATA)>
<!ELEMENT allocation_policy (#PCDATA)>
<!ELEMENT scheduling_algorithm (scheduling_tasks,scheduling_messages)>
<!ELEMENT scheduling_tasks (#PCDATA)>
<!ELEMENT scheduling_messages (#PCDATA)>
```

Table 6-10 – Configuration description DTD

The configuration data group the various simulation options that tune the software operation.

Table 6-11 resumes the configuration parameters relevant for each unit and the log contents output by each unit.

	Procedure expansion	Parameter determination	Task allocation	Scheduling
Configuration	• EC.	• Approach: net-centric or node-centric.	• Allocation policy.	• Scheduling algorithms for tasks and for messages.
Log	• Execution time of this unit. • List of expanded procedures.	• Execution time of this unit. • List of automatically generated parameters.	• Execution time of this unit. • List of allocated tasks.	• Execution time of this unit. • Result of scheduling • Macro-cycle. • Start-up time.

Table 6-11 – Configuration and logging for each simulator unit

A scenario mainly comprises the information about which data structures should be used for the definition of the architecture and entities, and also the configuration of a particular simulation. Therefore, a simulation scenario defines a set of data structures to be considered for a particular environment to be simulated. The DTD is presented in Table 6-12.

```
<!ELEMENT scenario (name,architecture,entities,configuration)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT architecture (#PCDATA)>
<!ELEMENT entities (#PCDATA)>
<!ELEMENT configuration (#PCDATA)>
```

Table 6-12 – Scenario description DTD

This way, architecture descriptions, and other documents, can be reused in several scenarios.

6.6.2.2 Resolving dependences between data streams

As explained in section 4.1.2, when a task participates in more than one data stream the calculated initial phase, Ph, for each data stream is, most probably, different. The solution proposed is based on an ordered task list. For example, the list that corresponds to the data streams shown in Figure 6-22 could be the one depicted in Figure 6-23.
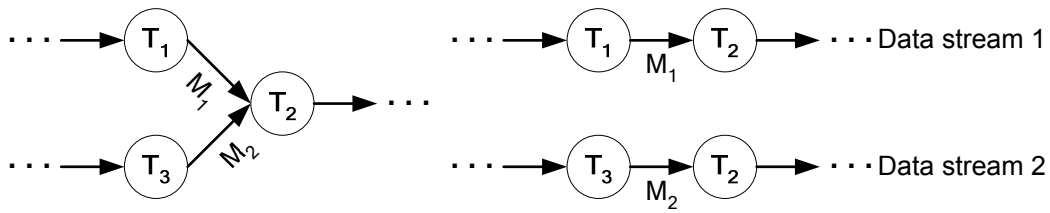
Figure 6-22 – Task interaction and correspondent data streams
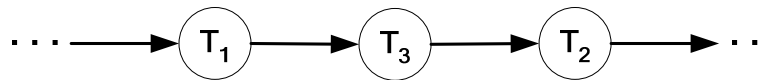
Figure 6-23 – Example of an ordered task list

Another possibility that is equivalent is to have task T_3 before task T_1.

An algorithm to accomplish this verification is presented in Figure 6-24.
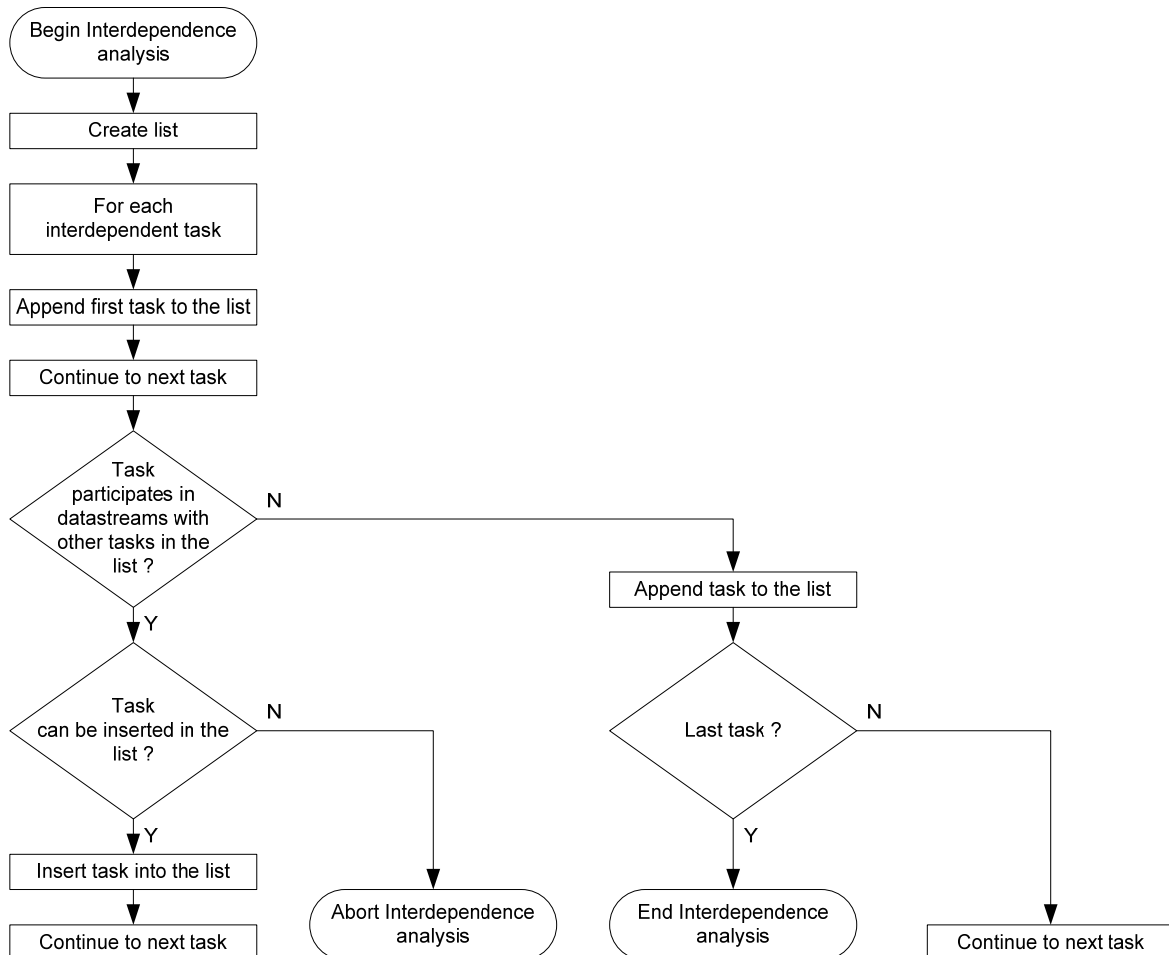
Figure 6-24 – Flowchart of the interdependence analysis

The most challenging step in the algorithm is checking if a task can be inserted in the ordered task list. This requires the analysis of the various data streams where the tasks, which are already in the list, participate.

6.6.3 Implementation

In terms of implementation, the main goals were:

- The use of the Java programming language;
- The use of XML files for all input and output data, including intermediate data that is exchanged between units;
- Increase the control of the simulation through configuration;
- Report unit operation through logging;
- Function operation as close as possible to an actual working system;
- Clear separation between the various units, allowing, in the future, an easy integration within a framework;
- The units function in batch mode, therefore, there is no integrated user interface;
- Exception handling, providing a good fault coverage resulting in a smooth operation.

The use of the Java language [Sun05] [Sim04] [HC05] in the second version of the SimHol was due to the wide support and portability. This implementation choice makes possible to execute the same simulator within different environments. For a particular environment, all that is required is the availability of a virtual machine with support for the classes that are used.

This version of the simulator implements both the net-centric approaches and the node-centric approaches, according to Chapter 1. It also implements the concept of a procedure, as explained in the same chapter.

One of the best features of this implementation is the integration with other tools. Due to the use of XML files and the output of all meaningful data, other tools can easily be attached to the simulator units, or even, any simulator unit can be independently integrated into another tool.

The development of the second version of the SimHol is being done with the Eclipse [Ibm03] [Car05] Integrated Development Environment (IDE).

6.6.4 Upgrading

Due to the use of the Java language, upgrading the SimHol has become easier.

If a new architecture is to be added, all that is required is a new class that inherits from the class Bus. This new class has to implement the method *public int msgSize (int nBytes)* that returns the number of bits necessary to transmit a message for the current architecture.

This class should be placed in the correct package and compiled. Due to the nature of the linking procedure in Java, which is accomplished in run-time, there is no need to update any other simulator file.

If a new algorithm is to be added and the needed criteria are already available then no much change is required. If the criterion is not available then it has to be added to each entity together with the correspondent getters and setters, and its value has to be manipulated accordingly.

Other upgrades are also possible and are not difficult due to the design choice of having separate units.

6.7 Future version

A future version could contemplate the following aspects:

- Allow the simulation of task admission during run-time and
- Allow the parameter change of tasks and messages.

7 A case study: the CAMBADA robots

This chapter addresses a specific case study that concerns the CAMBADA robots [SMA+05], developed at the University of Aveiro, for a participation at the RoboCup Middle Size League. These robots have a low level distributed sensing and actuation system based on Controller Area Network (CAN) that interconnects the motor drives, the movement controllers, the odometry system and other subsystems detailed later. This work focuses on the communication and synchronization of activities, which is carried out using the FTT-CAN [APF02] protocol. It is shown how to implement an application on top of this protocol as well as some of the benefits that arise from its use with respect to other communication alternatives based on non-globally synchronized frameworks. The communication requirements were also studied in [SFN+05]. In this first approach, empiric methods were used to define the deadlines and initial phases of tasks and messages so that, for each procedure, both the end-to-end delay and the jitter were reduced. As time goes by, the continuous development of this system may bring new functionalities, new nodes or upgraded computing power at existing nodes. When considering an environment with changing conditions and requirements, the empiric methods become more and more difficult to use because everything has to be redone from the ground. When using the approaches presented in Chapter 4, the system designer can guide the parameter determination in a flexible way without much tweaking.

This chapter is structured as follows. Section 7.1 introduces this type of systems and presents some related work. Section 7.2 shows the general architecture of the CAMBADA robots, while the respective communication and computation requirements are analyzed in section 7.2. Section 7.3 addresses some relevant implementation issues, mainly those concerning the use of the communication system and the synchronization of activities across the distributed system. Section 7.4 shows the benefits of the use of FTT-CAN with the CAMBADA robots. Section 7.5 shows the use of other approaches to the parameter determination while section 7.6 compares the three types of approach. The required buffering for messages is covered in section 7.7. Finally, the kernel of the nodes is commented in section 7.8.

7.1 Introduction

The control of robots, particularly autonomous mobile robots, is one of the application fields where Distributed Embedded Systems (DES) have been increasingly used, seeking for cabling reductions and simplification, improved maintainability, fault-tolerance and scalability of functionality.

As referred before, there are several advantages that may arise from the use of distributed architectures in embedded control systems and such a distributed approach has been often used in the specific field of mobile and autonomous robotics for diverse application scenarios. For example, [CSM97] presents a robot for orange picking that is divided into 4 platforms, each one with two picking arms. An SP50 (later Foundation Fieldbus FF-H1) fieldbus is used to provide connectivity between the four platforms and support the required data exchanges. [VSC$^+$99] presents an industrial robot based on a ProfiBus network. The authors simulate the system operation using Matlab/ Simulink, and measure the communication delays and level of synchrony achieved among the activities carried out within the robot.

One particular protocol that has been substantially used within mobile robots is CAN [Bos91] due to its low price, good reliability and timeliness properties. Examples of using this protocol can be found in [MN99], [KA02] [YPS$^+$02]. The latter one is particularly relevant to this work as it addresses the concerns of supporting a distributed sensing and actuation system integrated in a more complex architecture encompassing a deliberative level that extends beyond the robot. A TCP/IP connection with an adequate temporal firewall is used to isolate this level from the lower one in which real-time constraints are tight. In [PBB$^+$03] the authors discuss the impact that the communication jitter of real-time data transfers can have on the performance of control closed-loops and propose a mixed CAN-based event/time-triggered protocol.

The control architectures referred above either use event-triggered approaches that present poor control over the communication jitter, given the absence of relative offsets, or they use time-triggered approaches for the periodic traffic specified in a static way. In this work we address the issues arising from the use of FTT-CAN [APF02] to support the distribution of low level sensing and actuation information. This protocol provides support for flexible time-triggered communication, thus allowing to adapt the rates of the periodic communication on-line according to the instantaneous needs. [BCR$^+$99] shows the interest

of providing dynamic rate adaptation of the periodic information in a mobile robot but using a centralized architecture. Our work allows extending those benefits to a distributed framework.

7.2 General architecture

The general architecture of the CAMBADA robots has been described in [ASF[+]04]. Basically, the robots follow a biomorphic paradigm, each being centred on a main processing unit, the *brain*, which is responsible for the higher-level behaviour coordination, i.e. the coordination layer. This main processing unit handles external communication with the other robots and has high bandwidth sensors, typically vision, directly attached to it. Finally, this unit receives low bandwidth sensing information and sends actuating commands to control the robot attitude by means of a distributed low-level sensing/actuating system, the *nervous system* (Figure 7-1).
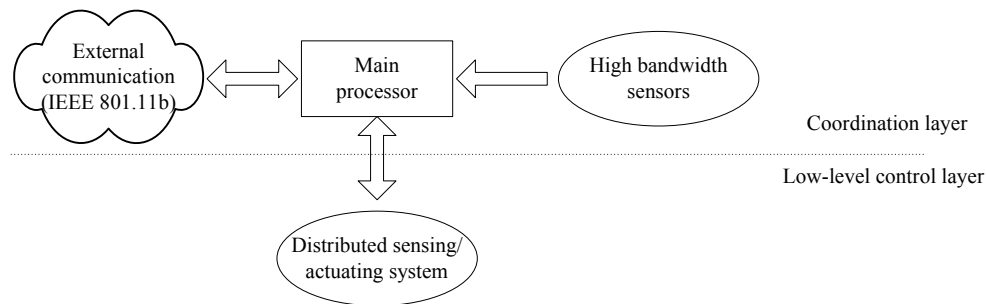
Figure 7-1 – The biomorphic architecture of the CAMBADA robots

At the heart of the coordination layer is the Real-Time Database (RTDB) that contains both the robot local state information as well as local images of a subset of the states of the other robots. A set of processes update the local state information with the data coming from the vision sensors as well as from the low-level control layer. The remote state information is updated by a process that handles the communication with the other robots via an IEEE 802.11b wireless connection. The RTDB is then used by another set of processes that define the specific robot behaviour for each instant, generating commands that are passed down to the low-level control layer (Figure 7-2).
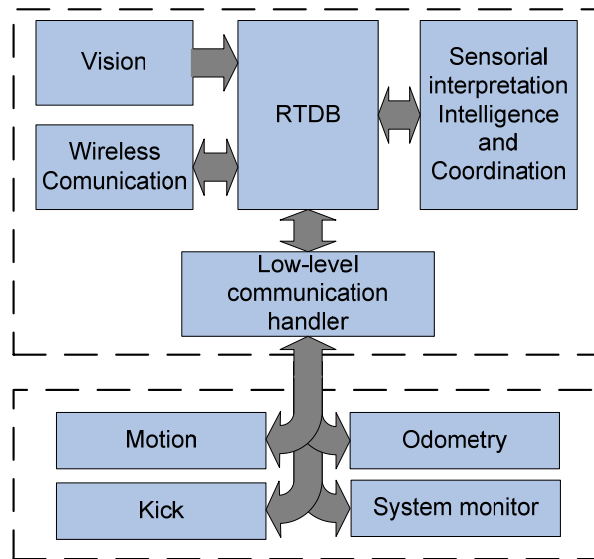
Figure 7-2 – Functional architecture of the robots built around the RTDB

The low-level sensing/actuating system follows the fine-grain distributed model [Kop97] where most of the elementary functions, e.g. basic reactive behaviours and closed-loop control of complex actuators, are encapsulated in small microcontroller-based nodes interconnected by means of a network. The nodes are based on the PIC microcontroller 18Fx58 [Pic05] operating at 40MHz while the network uses the CAN protocol with a bit rate of 250Kbps.

At this level there are 3 DC motors with respective controllers plus an extra controller that, altogether, provide holonomic motion to the robot. Each motor has an incremental encoder that is used to obtain speed and displacement information. Another node is responsible for combining the encoder readings from the 3 motors and for building coherent displacement information that is then sent to the coordination layer. Moreover, there is a node responsible for the kicking system that consists of a couple of sensors to detect the ball in position and trigger the kicker. This node also carries out battery voltage monitoring. Finally, the low-level control layer is interconnected to the coordination layer by means of a gateway attached to the serial port of the PC, configured to operate at 115Kbaud. From the perspective of the low-level control layer, the higher coordination layer is hidden behind the gateway and thus, we will refer to the gateway as the source or destination of all transactions arriving from or sent to that layer.

136

7.3 Lower-level requirements

In the previous section the functional and hardware architectures of the low-level control layer have been identified. The specific mapping of the former over the latter generates the operational architecture which presents requirements concerning both the tasks that need being executed on each node as well as the messages that must be exchanged over the network. In this section, these requirements, which were used for the actual implementation, will be analyzed in detail.

The *Motion* procedure depicted in Figure 7-2 spans across 4 nodes, the 3 motor controllers plus the holonomic controller that translates the robot velocity vector set-point received from the upper layer into individual speed set-points for each of the motors (see Figure 7-3). Both the motor controllers as well as the holonomic controller execute in a periodic fashion but with different periods. The former ones execute a PI-type closed-loop motor speed control once every 5ms. This value has been deduced from the dynamics of the robot. Moreover, these tasks are relatively light, taking less than 1 ms to accomplish. On the other hand, the holonomic controller executes a cyclic conversion of the higher layer set-points once every 30ms. This node is relatively loaded as each conversion takes about 16ms to carry out. The chosen period is, nevertheless, sufficiently small to support a smooth robot motion.
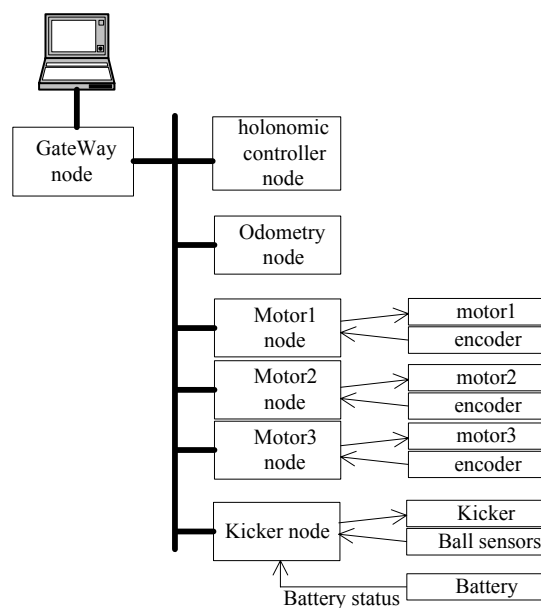
Figure 7-3 – Hardware architecture of the low-level control layer

In terms of communication, the *Motion* procedure requires the periodic transfer of the robot velocity vector set-point from the gateway to the holonomic controller and then the periodic transfer of the motor speed set-points from the holonomic controller to the individual motor controllers. Both transfers are carried out once every 30ms. The former transfer requires two messages (M6.1, M6.2) to convey the linear and angular information respectively. Concerning the latter transfer, the motor speed set-points generated for the motor controllers should be applied to each motor approximately at the same time. Thus they are piggybacked on the same message and transferred as a broadcast (M1). Finally, the control loops of the 3 motor controllers should also be synchronized among themselves so that they generate motor actuation signals at approximately the same time. Procedure Motion has a period property that is equal to 30ms. This procedure is represented in Figure 7-4.
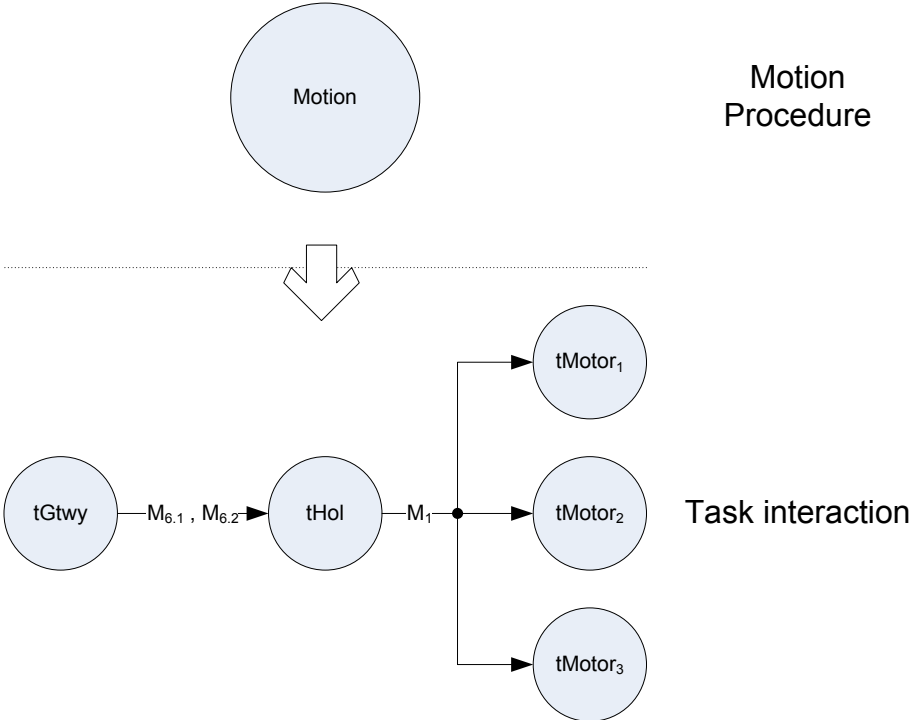
Figure 7-4 – The *Motion* procedure

Another important subsystem is the one corresponding to the *Odometry* procedure. This procedure also spans across 4 nodes, the 3 motor controllers plus a 4th node that combines the individual encoder readings into coherent displacement information sent up to the higher layer (see Figure 7-3). The encoder readings are the same as used by the closed-loop motor speed control and thus they are sampled every 5ms, and this should be carried out

synchronously in all three motors. However, depending on the desired precision in constructing the robot displacement information, these readings can be sent with a periodicity that varies from 5ms to 20ms (higher to lower precision). During the execution of certain high level behaviours the odometry information is not needed, e.g. when tracking the ball, and thus it can also be temporarily switched off. Three messages are used to convey the encoder readings (M3.1-M3.3). Upon reception of these messages, the odometry node calculates the robot position and orientation, taking approximately 4ms, and sends it to the higher layer, every 50ms, using 2 messages (M4.1, M4.2). This period is compatible with the cycles used by the processes running within the higher layer.

The *Odometry* procedure also includes a pair of sporadic messages (M5.1, M5.2) received from the higher layer to set or reset the current robot position and orientation information within the odometry node. These messages are not expected to be generated within less than 500ms intervals (minimum inter-arrival time – *mit*). Procedure Odometry has a period property that is equal to 30ms. This procedure is represented in Figure 7-5.
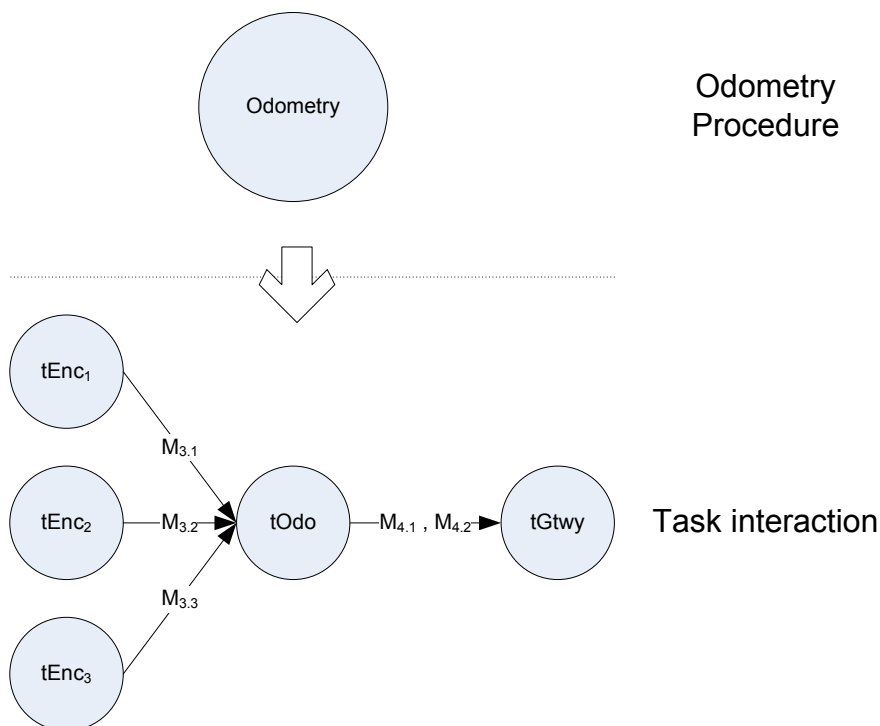
Figure 7-5 – The *Odometry* procedure

Finally, the *Kick* and *System monitor* functions are integrated in the same node, the kicker controller, which is lightly loaded. The former corresponds to executing the kicking commands received from the higher layer. These are conveyed within one sporadic

message (M7) which is not expected to be transmitted more often than once every second. In fact, the kicker is electromagnetic and takes about this time to recharge between consecutive kicks. On the other hand, the latter function currently encompasses the batteries level sampling which is sent up to the higher layer using a periodic message (M2) with a period of 1s, as well as a set of 5 sporadic messages (M8-M12) that inform the higher layer whenever a hard reset occurs in the respective node. A summary of the message roles is presented in Table 7-1.

Id	Type	Short description
M1	Periodic	Aggregate motor speeds set points
M2	Periodic	Battery status
M3.1-M3.3	Periodic	Wheels encoder values
M4.1-M4.2	Periodic	Robot position + orientation
M5.1-M5.2	Sporadic	Set/reset robot position + orientation
M6.1-M6.2	Periodic	Velocity vector (linear+angular)
M7	Sporadic	Kicker actuation
M8-M12	Sporadic	Node hard reset

Table 7-1 – Summary of message roles

The communication requirements are shown in Table 7-2. In this table, the maximum message size is shown.

Procedure	Id	Type	Size (Bytes)	CAN Msg Size (bits)	C - 125kbs (microsec)	T/mit (milisec)
Motion	M1	Periodic	6	99	774	30
-	M2	Periodic	2	59	461	1000
Odometry	M3.1	Periodic	3	69	540	10
Odometry	M3.2	Periodic	3	69	540	10
Odometry	M3.3	Periodic	3	69	540	10
Odometry	M4.1	Periodic	7	109	852	50
Odometry	M4.2	Periodic	4	79	618	50
-	M5.1	Sporadic	7	109	852	500
-	M5.2	Sporadic	4	79	618	500
Motion	M6.1	Periodic	7	109	852	30
Motion	M6.2	Periodic	4	79	618	30
-	M7	Sporadic	1	49	383	1000
-	M8	Sporadic	2	59	461	1000
-	M9	Sporadic	2	59	461	1000
-	M10	Sporadic	2	59	461	1000
-	M11	Sporadic	2	59	461	1000
-	M12	Sporadic	2	59	461	1000

Table 7-2 – Messages' parameters

The tasks involved in the procedures have their specifications shown in Table 7-3.

Procedure	Id	Node	Cons Msgs	Prod Msgs	C (milisec)	T (milisec)
Odometry	tEnc1	Motor1	-	0 , M3.1	1	5
Odometry	tEnc2	Motor2	-	0 , M3.2	1	5
Odometry	tEnc3	Motor3	-	0 , M3.3	1	5
Odometry	tOdo	Odometry	(M3.1,M3.2,M3.3)	0 , (M4.1,M4.2)	4	10
Odometry	tGtwyOdo	GateWay	(M4.1,M4.2)	-	(4)	50
Motion	tGtwyHol	GateWay	-	(M6.1,M6.2)	?	Event
Motion	tHol	Holonomic	(M6.1,M6.2)	M1	16	30
Motion	tMotor1	Motor1	M1	-	1	30
Motion	tMotor2	Motor2	M1	-	1	30
Motion	tMotor3	Motor3	M1	-	1	30

Table 7-3 – Tasks' parameters

The task tGwtyOdo, which is responsible for transmitting odometric data to an external computational unit through the serial port, has an estimated execution time of 4ms. On the other hand, the execution time of task tGtwyHol could not be estimated because it is dependent on the rate at which data is received through the serial port. This task has an asynchronous behaviour. Due to these properties, it is understood that its execution will be scattered through its execution window.

7.4 Low-level control layer implementation

After having defined the operational architecture of the low-level control layer and deduced the computing and communication requirements the practical implementation was carried out. Two approaches have been followed, one without and another with global synchronization among the activities executed at this layer. The former approach used communication functions of the type *send* and *receive*, as commonly found in event-triggered systems, and without any further support for synchronizing remote activities. At fixed points within the respective cycle the data would be transmitted using the *send* function and retrieved at the receiver with the *receive* function.

The temporal behaviour of the approach referred above may suffer large delays due to the multiple unsynchronized chained cycles. For example, consider that a new velocity vector arrived at the holonomic controller right after it started processing one cycle. Then, the new vector would be processed one cycle later, generating speed set-points for the motors with about 30ms additional delay, i.e., the cycle time of the holonomic controller. These

set-points would then be transmitted over the network within one message, possibly suffering an access delay caused by possible transmissions from other unsynchronized nodes. Finally, this message would arrive at a motor node right after this node had started one speed control cycle thus holding the new set-point until the next cycle causing a further delay of 5ms. Comparing with the case in which the new data would arrive just before the start of the cycle in which it would be used, i.e. the best-case delay, the previous situation corresponds to an additional delay of more than 35ms to process a new velocity vector. Figure 7-6 illustrates the impact of chained non-synchronized cycles on the end-to-end delay (d_{ee}) for the general case of two periodic tasks in different nodes, A and B, which communicate via a periodic message.
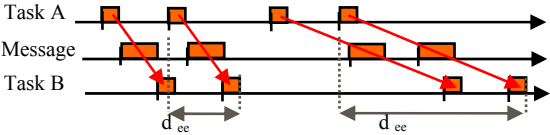
Figure 7-6 – Synchronization and end-to-end delay

Moreover, this delay can vary on-line due to drifts in the local clocks of the nodes, generating jitter in the control signals. These additional delays and jitter can cause degradation to the global control loops associated to high level behaviours, such as tracking the ball.

On the other hand, this non-synchronized approach has the advantage of being very simple to deploy. For this reason, it was the first approach to be implemented. As expected, the parameters of the global control loops were relatively difficult to tune and a "nervous robot" behaviour was frequently observed.

Therefore, it was decided to use a communication infrastructure based on CAN that would allow building a globally synchronized framework so that relative phases among all activities in the system, including tasks execution in the nodes and message transfers over the network, could be established as appropriate to maintain the end-to-end delays of the information flows under tight bounds.

In order to use FTT-CAN two more nodes were added to the low-level control layer to perform the Master function with replication for fault-tolerance purposes [MFA+02] [FAM+03].

7.4.1 Implementation using FTT-CAN

In order to effectively use FTT-CAN in a given application it is necessary to identify the flows of information related with cyclic activities executed in the system, to determine what triggers each of those flows and then to determine which should be the appropriate offset of each transmission or activity knowing the respective transmission and execution times. This has been carried out in the previous section where the information flows within the low-level control layer of the CAMBADA robots were identified and characterized. In this section it will be shown how FTT-CAN was used to support the required synchronization.

The first aspect is to separate the periodic from the sporadic traffic. The latter is handled by the asynchronous subsystem similarly to a non-synchronized framework. This separation is already accomplished in Table 7-1. The periodic traffic is then named using FTT-CAN synchronous identifiers.

The EC duration is set to 5ms which is the shortest period among all periodic activities and messages, i.e., the closed-loop motor speed control period. For a trigger message with 5 bytes, the communication overhead is lower than 8.4% (420μs/5ms) while the computing overhead is close to 1.5% (76ms/5ms). These values were considered admissible given the application load. Particularly, the communication load according to Table 7-2 is close to 27% of the bus bandwidth at 250Kbps.

Knowing the EC duration, all periods are expressed in number of ECs. Then, the synchronization requirements are analyzed to identify the set of activities that needed synchronization and the respective set of synchronous triggers. These inherit periods equal to those of the related messages and are also named using appropriate FTT-CAN identifiers.

For each task or message, the deadline is considered to be equal to the period. This indicates that there are no particular timing requirements apart from having the execution windows independent from the transmission windows for consecutive instances of a task, or message.

Finally, the initial phases (off-sets) of all messages and synchronous triggers are established so that transmissions are carried out soon after the respective data becomes available and, conversely, activities are triggered enough in advance to generate data before but as close as possible to the respective transmission instant. Moreover, the

synchronous triggers allow triggering several remote activities at approximately the same time (within a few micro-seconds), as it is required by the *Odometry* procedure. These concerns lead to increase the freshness of the data in the information flows, reducing the respective end-to-end latency and jitter, with a positive impact in the performance of the respective global control loops associated to the high level behaviours.

FTT-CAN ID	Source	Destination	Period (#ECs)	Init time (#ECs)	Short description
0	Holonomic contr	Motor node[1:3]	6	5	Motors speed setpoints
1	Motor 1 node	Odometry node	2 (0-4)	2	Encoder Count in motor 1
2	Motor 2 node	Odometry node	2 (0-4)	2	Encoder Count in motor 2
3	Motor 3 node	Odometry node	2 (0-4)	2	Encoder Count in motor 3
4	Odometry node	Gateway	10	4	Current position
5	Odometry node	Gateway	10	4	Current orientation
6	Gateway	Holonomic contr	6	0	Velocity vector (linear)
7	Gateway	Holonomic contr	6	0	Velocity vector (angular)
8	---	Motor node[1:3]	1	0	Triggers the encoder readings
9	---	Motor node[1:3]	2	1	Triggers production of messages 1,2,3 at the motor nodes (encoder readings)
10	---	Odometry node	2	3	Triggers the consumption of encoder messages 1,2,3 at the odometry node
11	---	Odometry node	10	3	Event to produce messages 4,5
12	---	Holonomic contr	6	1	Triggers the consumption of Messages 6,7 in holonomic controller
13	---	Motor nodes [1:3]	6	6	Triggers the consumption of Message 0 in the motor nodes

Table 7-4 – Synchronous requirements table

Table 7-4 shows the system synchronous requirements table (SRT), including both synchronous messages and triggers. The off-sets extracted from the system requirements are expressed in the column *init time* and they are also expressed in number of ECs.
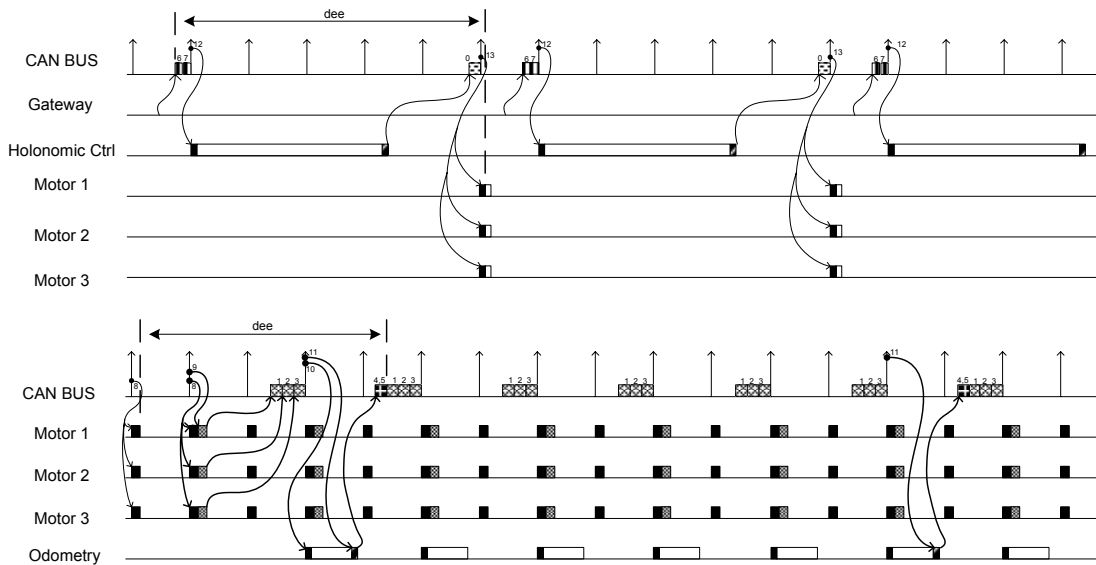
Figure 7-7 – Timeline of the main information flows within the low-level control layer

Figure 7-7 shows the timeline of the two main synchronous information flows, separately, associated to the *Motion* procedure (top) and the *Odometry* procedure (bottom). In what concerns the *Motion* procedure, the flow is triggered by a pair of messages, 6 and 7, sent by the gateway with off-set 0 and arriving from the higher layer with a velocity vector. These values are received by the holonomic controller that is synchronized in the EC started with trigger message 12, which is produced right after the transmission of the messages, with off-set of 1 EC. This trigger message starts the execution of the holonomic controller to process the new velocity vector. The resulting motor speed set-points will be available after 16ms, which rounds up to 4 ECs. Thus the respective message (0) is transmitted to the motor nodes in the following cycle, i.e. with an off-set of 5 ECs. Trigger message 13 is used to synchronize the closed-loop speed control of each motor with the arriving set-point. The off-set is 6 ECs to enforce a reduced latency between reception and use of the set-points.

The transmission of the next velocity vector, and thus the start of the next cycle, is carried out in the following EC.

In what concerns the *Odometry* procedure, the respective information flow starts with trigger message 8, with off-set 0, which causes the synchronous sampling of the encoders in the 3 motors. These values are locally accumulated until they are transmitted. In the example, the transmission of the encoder readings is set to 2 ECs (messages 1-3) and the respective values are produced with trigger message 9, in the EC before their transmission.

Thus the off-set of messages 1-3 is 2 ECs while the off-set of trigger message 9 is 1 EC. The periods of these entities can vary depending on the desired odometry precision from 1 EC (highest) to 4 ECs (lowest). They can also be suspended (period set to 0) when the *Odometry* procedure is not needed.

7.5 Other approaches to the parameter determination

In the previous sections, the benefits of the use of FTT-CAN with the CAMBADA robots were established. In this first approach, empiric methods were used to define the deadlines and initial phases of tasks and messages so that, for each procedure, both the end-to-end delay and the jitter were reduced. As time goes by, the continuous development of this system may bring new functionalities, new nodes or upgraded computing power at existing nodes. When considering an environment with changing conditions and requirements, the empiric methods become more and more difficult to use because everything has to be redone from the ground. When using the approaches presented in Chapter 4, the system designer can guide the parameter determination in a flexible way without much tweaking. These approaches will now be used and a practical comparison is done.

7.5.1 Net-centric approaches

The net-centric approaches comprehend two techniques: the message deadline (MD) and the message maximum finishing (MMF). By using the equations derived in Chapter 4, new tasks' and messages' parameters were calculated. These are presented, respectively in Table 7-5 and Table 7-6.

| | | | | | Net-centric | | | |
| | | | | | Msg Deadline | | Msg Max Fin | |
Procedure	Id	Node	C (milisec)	T (milisec)	D (milisec)	Ph (milisec)	D (milisec)	Ph (milisec)
Odometry	tEnc1	Motor1	1	5	5	0	5	0
Odometry	tEnc2	Motor2	1	5	5	0	5	0
Odometry	tEnc3	Motor3	1	5	5	0	5	0
Odometry	tOdo	Odometry	4	10	5	10	5	10
Odometry	tGtwyOdo	GateWay	(4)	50	5	60	5	20
Motion	tGtwyHol	GateWay	?	Event	-	-	-	-
Motion	tHol	Holonomic	16	30	20	25	20	5
Motion	tMotor1	Motor1	1	30	5	0	5	0
Motion	tMotor2	Motor2	1	30	5	0	5	0
Motion	tMotor3	Motor3	1	30	5	0	5	0

Table 7-5 – Tasks' parameters resulting from the net-centric approaches

It can be seen that there are no differences between the deadlines calculated using both methods. This can be explained considering the nature of the net-centric approaches, where the transmission windows are favoured and the execution windows are kept to the minimum possible. It is seen that the initial phases are smaller in the second approach. This is due to the two phase process that tries to reduce as much as possible the transmission windows of the involved messages.

| | | | | | Net-centric | | | |
| | | | | | Msg Deadline | | Msg Max Fin | |
Procedure	Id	Type	C - 125kbs (microsec)	T/mit (milisec)	D (milisec)	Ph (milisec)	D (milisec)	Ph (milisec)
Motion	M1	Periodic	774	30	25	45	5	25
Odometry	M3.1	Periodic	540	10	5	5	5	5
Odometry	M3.2	Periodic	540	10	5	5	5	5
Odometry	M3.3	Periodic	540	10	5	5	5	5
Odometry	M4.1	Periodic	852	50	45	15	5	15
Odometry	M4.2	Periodic	618	50	45	15	5	15
Motion	M6.1	Periodic	852	30	25	0	5	0
Motion	M6.2	Periodic	618	30	25	0	5	0

Table 7-6 – Messages' parameters resulting from the net-centric approaches

From the side of the messages' parameters, the MD approach widens the transmission window as much as possible while guaranteeing that the message will be consumed before the following instance can be produced. This is why the deadlines are equal to the period minus the value of one EC. Again, when comparing to the MMF approach, the initial

phases tend to be smaller. Also the deadlines are reduced to the minimum required so that they are still met.

The resulting timeline for the MD approach is shown in Figure 7-8. While the timeline for the MMF approach is shown in Figure 7-9.

Figure 7-8 – Timeline for the net-centric MD approach

Figure 7-9 – Timeline for the net-centric MMF approach

From the two timelines, it can be seen that the end-to-end delay is much smaller with the MMF approach, 5 ECs for the Odometry and 7 ECs for the Holonomic, than with the MD approach, 15 ECs for both the Odometry and the Holonomic.

7.5.2 Node-centric approaches

The node-centric approaches comprehend two techniques: the task deadline and the task maximum execution. By using the equations derived in Chapter 4, new tasks' and messages' parameters were calculated. These are presented, respectively in Table 7-7 and Table 7-8.

					Node-centric			
					Task Deadline		Task Max Fin	
Procedure	Id	Node	C (milisec)	T (milisec)	D (milisec)	Ph (milisec)	D (milisec)	Ph (milisec)
Odometry	tEnc1	Motor1	1	5	5	0	5	0
Odometry	tEnc2	Motor2	1	5	5	0	5	0
Odometry	tEnc3	Motor3	1	5	5	0	5	0
Odometry	tOdo	Odometry	4	10	5	10	5	10
Odometry	tGtwyOdo	GateWay	(4)	50	45	20	5	20
Motion	tGtwyHol	GateWay	?	Event	-	-	-	-
Motion	tHol	Holonomic	16	30	25	5	20	5
Motion	tMotor1	Motor1	1	30	5	0	5	0
Motion	tMotor2	Motor2	1	30	5	0	5	0
Motion	tMotor3	Motor3	1	30	5	0	5	0

Table 7-7 – Tasks' parameters resulting from the node-centric approaches

From the side of the tasks' parameters, the MD approach widens the execution window as much as possible while guaranteeing that the message will be consumed before the following instance can be produced. This is why the deadlines are equal to the period minus the value of one EC. An exception occurs with the tasks tEnc because they only produce a message on alternate ECs, in this case the deadline is equal to the period of the production minus the value of one EC. In this case it did not occur but, when comparing to the MMF approach, the initial phases tend to be smaller. Also the deadlines are reduced to the minimum required so that they are still met.

					Node-centric			
					Task Deadline		Task Max Fin	
Procedure	Id	Type	C - 125kbs (microsec)	T/mit (milisec)	D (milisec)	Ph (milisec)	D (milisec)	Ph (milisec)
Motion	M1	Periodic	774	30	5	30	5	25
Odometry	M3.1	Periodic	540	10	5	5	5	5
Odometry	M3.2	Periodic	540	10	5	5	5	5
Odometry	M3.3	Periodic	540	10	5	5	5	5
Odometry	M4.1	Periodic	852	50	5	15	5	15
Odometry	M4.2	Periodic	618	50	5	15	5	15
Motion	M6.1	Periodic	852	30	5	0	5	0
Motion	M6.2	Periodic	618	30	5	0	5	0

Table 7-8 – Messages' parameters resulting from the node-centric approaches

It can be seen that there are no differences between the deadlines calculated using both methods. This can be explained considering the nature of the node-centric approaches,

where the execution windows are favoured and the transmission windows are kept to the minimum possible. It is seen that the initial phases are smaller in the second approach. This is due to the two phase process that tries to reduce as much as possible the execution windows of the involved tasks.

The resulting timeline for the MD approach is shown in Figure 7-10. While the timeline for the MMF approach is shown in Figure 7-11.
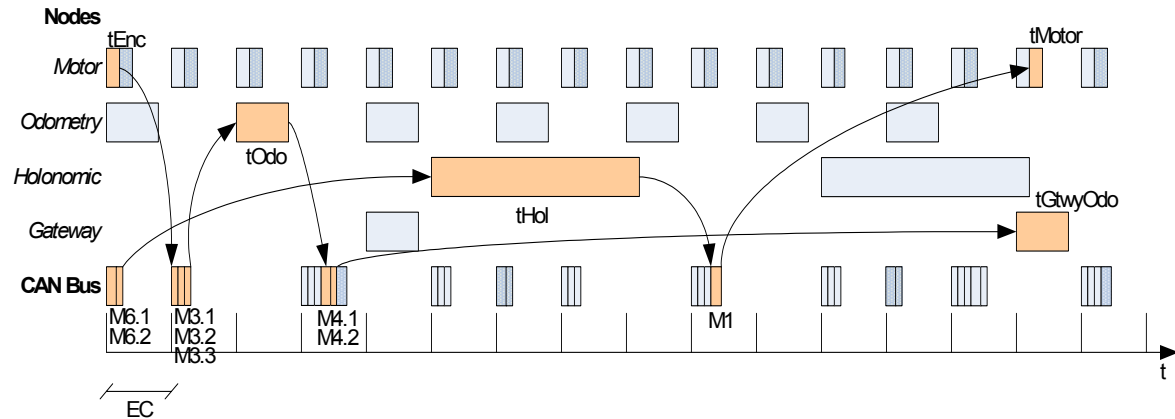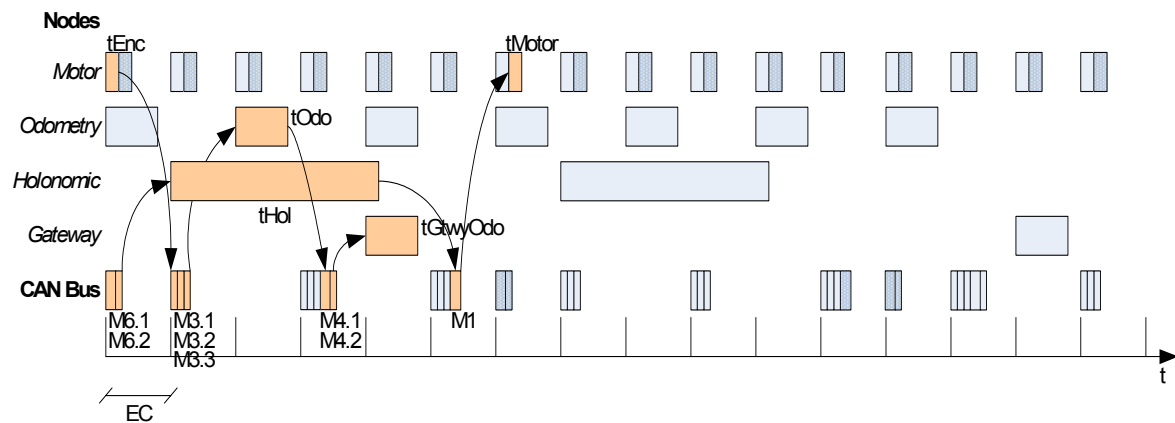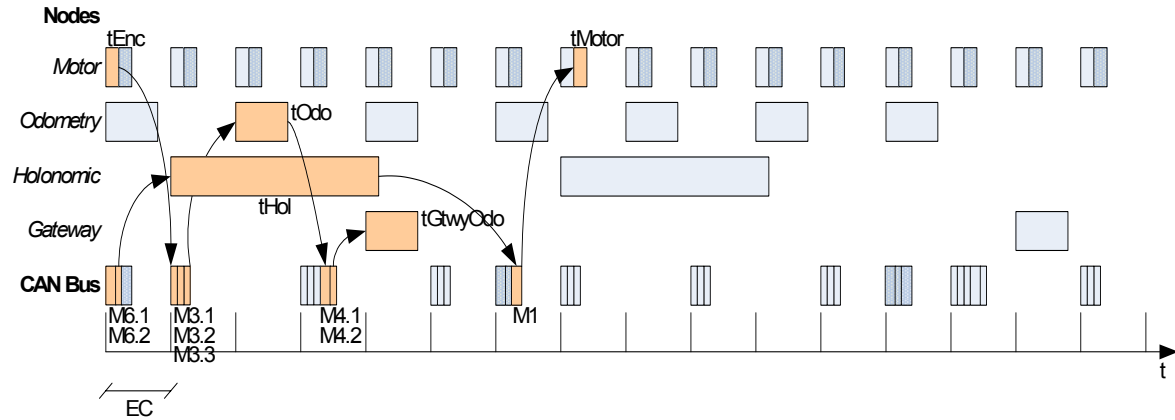

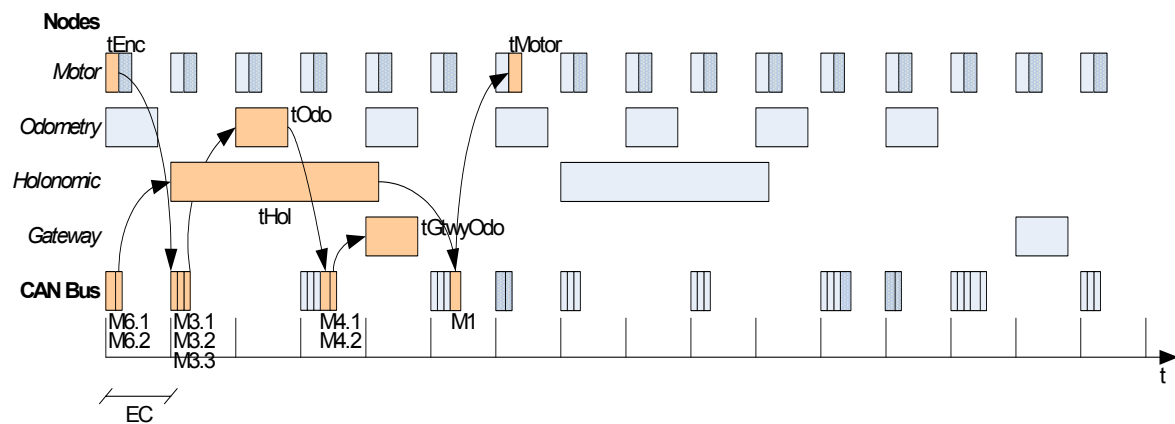**Figure 7-10 – Timeline for the node-centric TD approach**

Figure 7-11 – Timeline for the node-centric TMF approach

From the two timelines, it can be seen that the end-to-end delay is smaller with the TMF approach, 5 ECs for the Odometry and 7 ECs for the Holonomic, than with the TD approach, 5 ECs for the Odometry and 8 ECs for the Holonomic.

7.5.3 Overlapping approach

By using the equations derived in Chapter 4 for the overlapping approach, new tasks' and messages' parameters were calculated. These are presented, respectively in Table 7-9 and Table 7-10.

Procedure	Id	Node	C (milisec)	T (milisec)	Overlapping	
					D (milisec)	Ph (milisec)
Odometry	tEnc1	Motor1	1	5	5	0
Odometry	tEnc2	Motor2	1	5	5	0
Odometry	tEnc3	Motor3	1	5	5	0
Odometry	tOdo	Odometry	4	10	5	10
Odometry	tGtwyOdo	GateWay	(4)	50	45	20
Motion	tGtwyHol	GateWay	?	Event	-	-
Motion	tHol	Holonomic	16	30	30	5
Motion	tMotor1	Motor1	1	30	5	0
Motion	tMotor2	Motor2	1	30	5	0
Motion	tMotor3	Motor3	1	30	5	0

Table 7-9 – Tasks' parameters resulting from the overlapping approach

Procedure	Id	Type	C - 125kbs (microsec)	T/mit (milisec)	Overlapping	
					D (milisec)	Ph (milisec)
Motion	M1	Periodic	774	30	25	25
Odometry	M3.1	Periodic	540	10	5	5
Odometry	M3.2	Periodic	540	10	5	5
Odometry	M3.3	Periodic	540	10	5	5
Odometry	M4.1	Periodic	852	50	45	15
Odometry	M4.2	Periodic	618	50	45	15
Motion	M6.1	Periodic	852	30	25	0
Motion	M6.2	Periodic	618	30	25	0

Table 7-10 – Messages' parameters resulting from the overlapping approach

The overlapping approach produces the lowest possible initial phases when using FTT-CAN. This happens because produced messages are scheduled to be transmitted in the EC after its production and consumer tasks are scheduled to be executed in the EC right after the one where the message to be consumed is transmitted. Also the deadlines are extended

to the maximum that is possible. The trade-off for this flexibility is an increased jitter for tasks and messages. This naturally results from the dependency between the release instants of consecutive entities in a data stream.

The resulting timeline for the overlapping approach is shown in Figure 7-12.
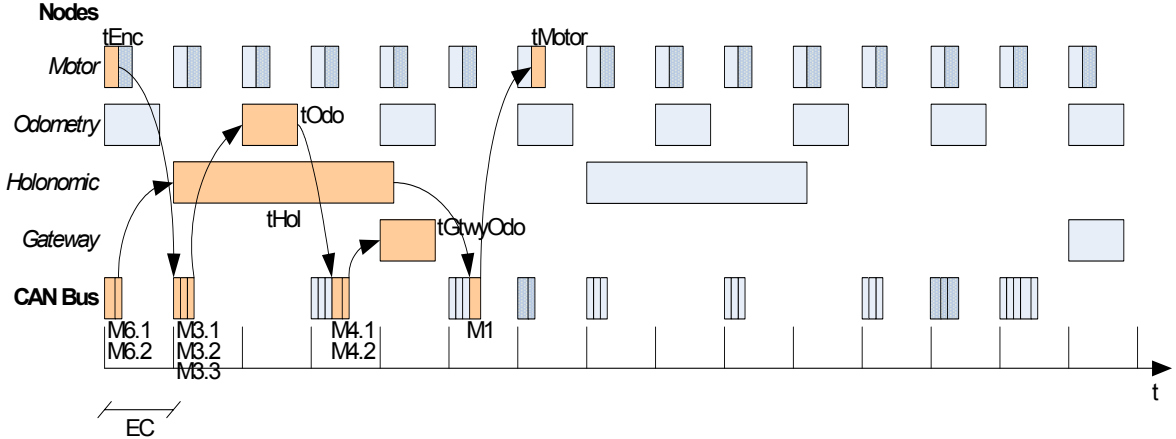
Figure 7-12 – Timeline for the overlapping approach

7.6 Comparing the three types of approach

For this case study, it was seen that a solution without jitter was found. But in an online system, with changing conditions and requirements the same cannot be guaranteed. The approaches cannot also give guarantees about jitter but they can reduce it in an automated way, which can be integrated into an admission control system. If the maximum jitter is known beforehand the use of the proposed approaches can help in admitting system changes that do not make the jitter too high, while rejecting the others.

As expected the Message, or Task, Maximum Finishing approach gave the best results in terms of the end-to-end delay. In online systems, this approach might require a recalculation every time a system change occurs. If this becomes unacceptable, then the Message, or Task, Deadline approach might be better. This approach has the worse end-to-end delay but for many applications this is not as important as the jitter. This approach gives better results in a changing environment because, due to the extended transmission or execution windows the system can accommodate extra load without a significant impact on the jitter. This is justified because this approach offers isolation between the execution and transmission windows of consecutive entities in a data stream.

Overall, the non-overlapping approaches, net-centric and node-centric, offer a lower jitter than the overlapping approach.

152

7.7 Buffering

After deriving the tasks' and messages' parameters, the buffering requirements for the transmission of messages can be calculated. The size of buffering, in bytes, for each EC and for each approach are presented in Table 7-11.

Transmission buffer (bytes)

Net-Centric	Message Deadline	9	0	26	0	9	0	9	0	15	0	9	0	20	0	15	0
	Message Maximum Finishing	9	0	26	0	15	0	9	0	9	0	15	0	20	0	9	0
Node-Centric	Task Deadline	9	0	20	0	9	6	9	0	9	0	9	6	20	0	9	0
	Task Maximum Finishing	9	0	20	0	15	0	9	0	9	0	15	0	20	0	9	0
	Overlapping	9	0	20	0	15	0	9	0	9	0	9	6	20	0	9	0
	EC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 7-11 – Buffering requirements for transmission

These values refer to the timelines presented for each approach. This process must be used for the entire macro-cycle so that the maximum requirements can be known. Just by looking at the table, which is just a snapshot of the macro-cycle, it can be understood that the net-centric approaches are more demanding in terms of memory required for buffering. The determination of the required size of memory can be important in systems with low resources where a choice has to be made between approaches with different buffering requirements.

7.8 Kernel of the nodes

The kernel functionalities currently implemented in the station nodes of the CAMBADA robots are depicted in Figure 7-13 as non-shaded areas, while the shaded areas represent future development.
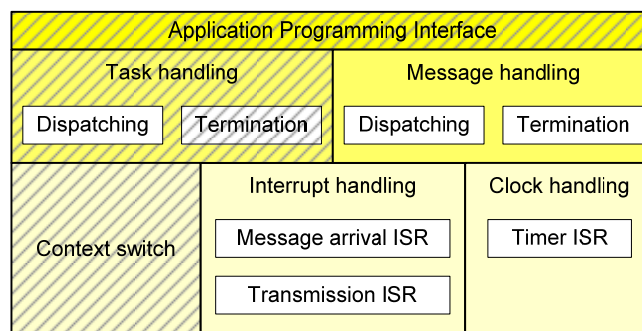
Figure 7-13 – Station kernel of the CAMBADA

Beginning with the lower level functionalities, the clock handling is being accomplished by an Interrupt Service Routine (ISR) that is the *Timer ISR* and the remaining interrupts are handled by the *Message arrival ISR* and the *Transmission ISR*.

The Timer interrupt occurs whenever the counter is started and counting reaches zero. Upon this the Timer ISR is called to handle the event (see Figure 7-14). This ISR reprograms the Timer in a way that it can be called in the beginning of each window within an EC. These windows are: the asynchronous window, the guardian window and the synchronous window. Then in dependence of the window, the Timer ISR interacts with a higher layer service that is the dispatching of messages, synchronous and asynchronous. This routine also interacts with the message handing service, namely with its dispatching routine.
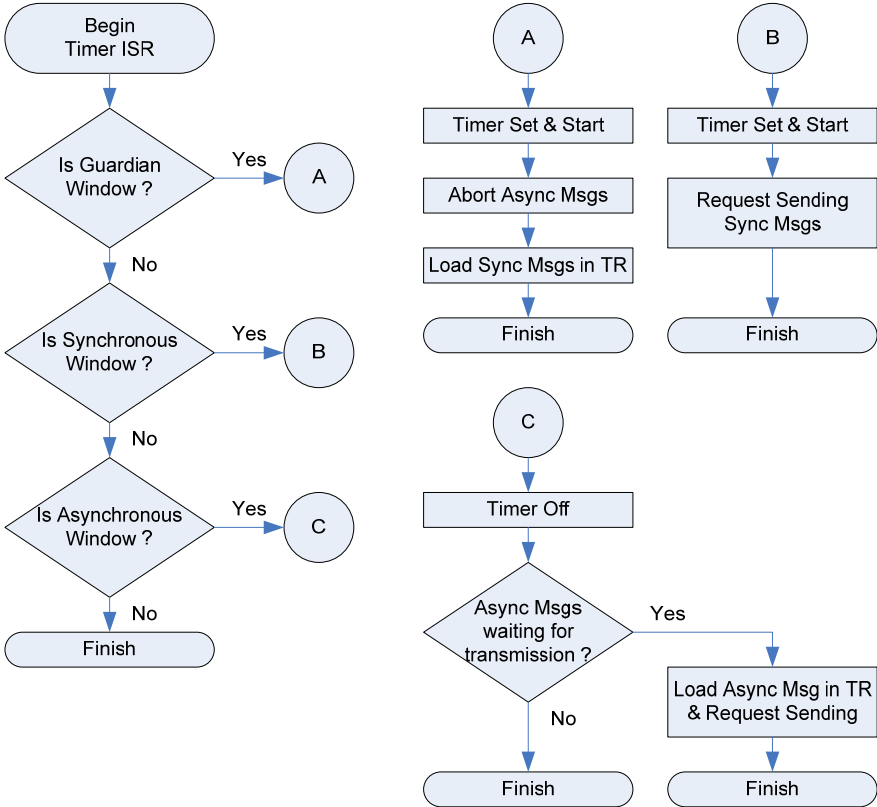
Figure 7-14 – CAMBADA: Timer ISR

A typical use of the clock is to set a time-slice. But in this context the time-slice that regulates the context switch is defined by the arrival of a TM so it is equal to the EC duration. The reason for this is that it is the Trigger Message TM that dispatches a task in the current EC. If a task is dispatched when another task has not finished execution it

154

means that either, this task has a lower priority, or that the higher priority task that is being executed will finish on this EC and the remaining time can be used to start the new task. So, if a higher priority task is ordered to be dispatched by a TM and another task is still in execution, then this task must be suspended so that the higher priority task can be dispatched. Later on, when this higher priority task finishes the suspended task can be resumed.
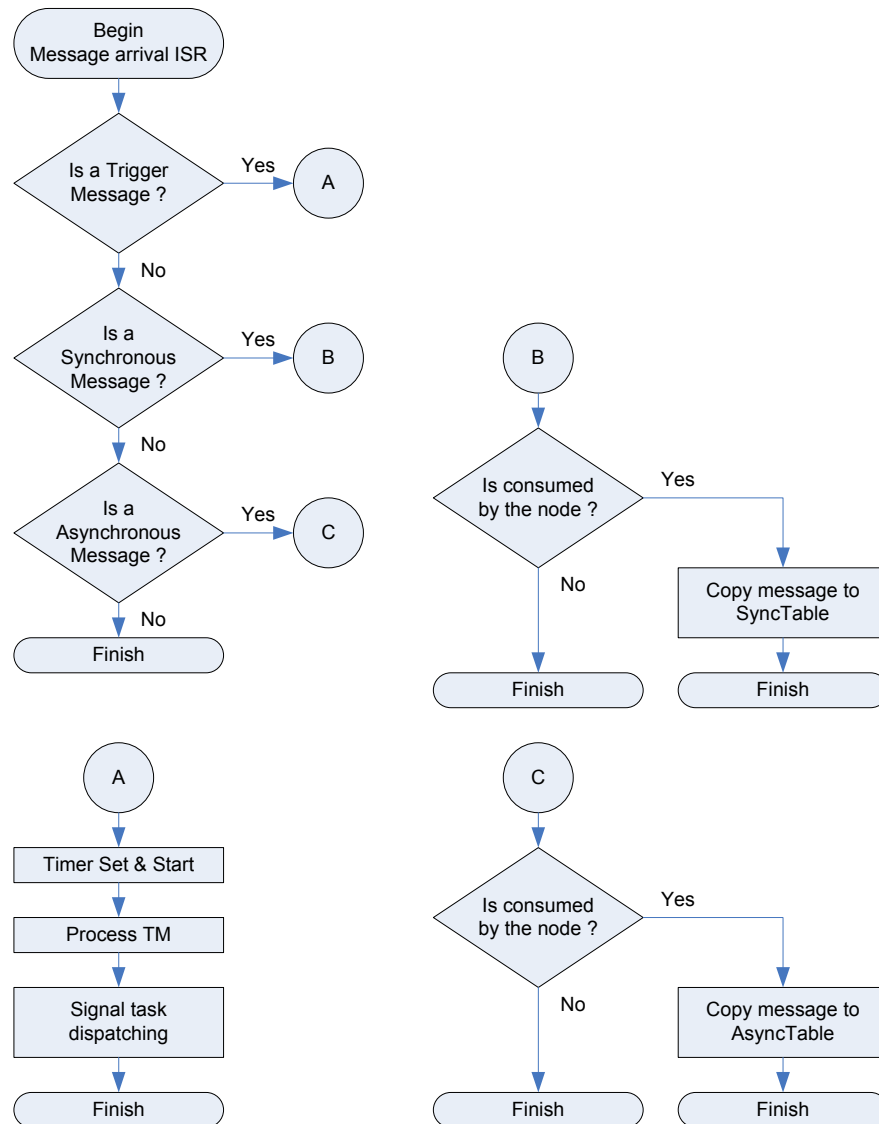
Figure 7-15 – CAMBADA: Message arrival ISR

When a message arrives at the node, the CAN controller issues a Message arrival interrupt. Upon this, the *Message arrival ISR* is called to handle the event (see Figure 7-15)). This ISR begins by checking the type of the received message that can be a trigger message, a synchronous message or an asynchronous message. Then if it is a TM it starts the timer,

checks which messages and tasks must be dispatched in the current EC and signals a task to be dispatched. If on the contrary, it is a synchronous message or an asynchronous message it just copies it to the respective buffer (table). This routine also interacts with the task dispatching.

When a message is transmitted by the CAN controller to the bus it issues a Transmission interrupt. Upon this, the Transmission ISR is called to handle the event (see Figure 7-16). This ISR checks which transmission window is active at the moment, either the asynchronous window or the synchronous window, and eventually tries to send the next message. This routine also interacts with the message handling service, namely with its termination routine.
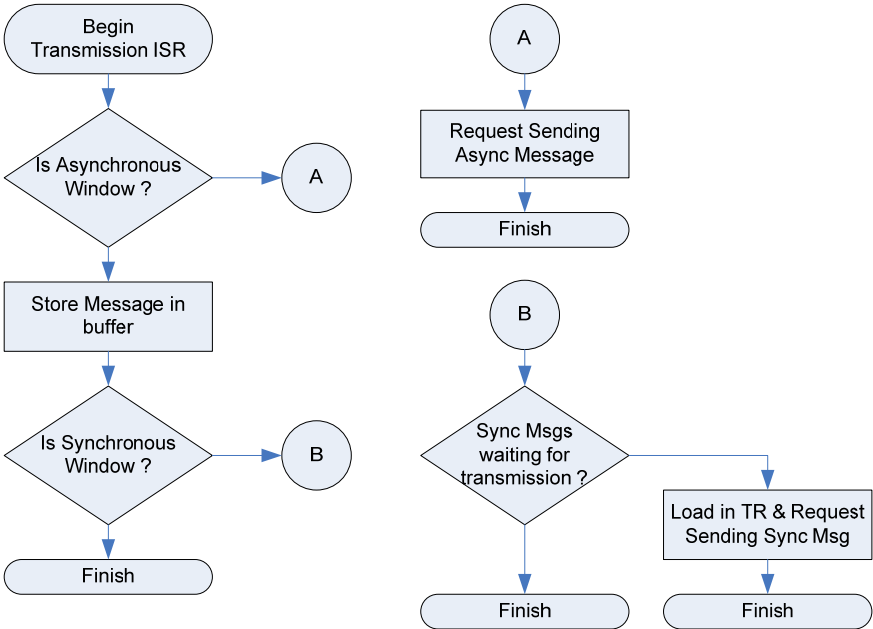
Figure 7-16 – CAMBADA: Transmission ISR

The middle layer functionalities are the task handling and the message handling.

Currently the task handling is only able to dispatch a task per EC and no other task can be running because its context could not be saved for later execution. Also a task is signalled for dispatching by embedding a type of polling mechanism into the task itself so that it can know when it is the right time to begin execution. In the future, this mechanism should be evolved to a more standard approach of having tasks independent of the kernel. On the other hand, the termination routine is not yet implemented. This routine would allow using the remaining free processing time on an EC to start another task that was previously instructed to be dispatched.

156

The message handling interacts with the CAN controller loading messages into its registers, requesting the sending and aborting messages already instructed to be sent. It should be noted that the actual sending of a message is under the control of the CAN hardware, so the kernel can only request that a message be sent. Aborting the messages involves invalidating the transmission buffer of the CAN hardware.

The upper layer is the Application Programming Interface and was not yet developed. This layer should offer basic services to the tasks isolating them from the hardware and the kernel features. A common benefit from this approach is that, after a good definition of the interface, the kernel can be changed without affecting the tasks. Therefore, this layer is of utmost importance.

7.9 Conclusions

In this chapter, a case study was used to demonstrate the automation of the proposed approaches for the parameter determination. The implementation of architectural proposals in the case study, like the joint dispatching of tasks and messages, the buffering of messages and the nano-kernels, was also discussed.

New criteria for the parameter determination were proposed. The criteria were based on the awareness on which system resource, either the communication or the computational infrastructures, is more constrained and the overlapping of the transmission and execution windows. The proposed criteria were shown to be a suitable basis for the bounding of the jitter of tasks and messages and for the control of the end-to-end delay.

The solution for the joint dispatching of tasks and messages using the FTT-CAN protocol was also shown to be a simple but effective one.

A partial implementation of the nano-kernel has also demonstrated that its simplicity is an answer to systems with low processing power nodes [CSFM06]. This kernel can be integrated in the computing nodes of a distributed control system without introducing a significant overhead. A central node dispatches both tasks and messages to other nodes that are only required to possess a nano-kernel, without a scheduler or a dispatcher. This approach shows how lightweight kernels can be used in computing nodes using low processing power microcontrollers.

An additional advantage of this solution is that, from a system point of view, it is possible to synchronize easily tasks in different nodes without incurring in additional overhead. In

the CAMBADA example presented all the motor nodes must acquire the encoder values at the same time. Using this kernel the synchronization of these tasks is made with relative high precision.

8 A platform independent software architecture

An embedded system integrates, basically, a processing unit, memory and I/O connections. Due to the always increasing computational power of embedded processors and significant memory sizes, it is now reasonable to expand the software layer to include a Java Virtual Machine (JVM).

One of Java's strengths is that it can run on any machine for which there is a compliant JVM. Every Java program, when it is compiled, gets translated into the same computer code. Every different hardware/operating system set has a program - the JVM - that can interpret between this universal Java computer code and the more case-specific parts of the software and hardware. This means that the exact same Java code can run on any computer system.

An open issue in these systems is still the timeliness guarantee. Due to the way the garbage collectors work it is not possible, in a standard Java system, to know the execution time of the programs, making it unusable for real-time applications. But an effort for the development of JVMs that can guarantee timeliness is going on. Currently, at least, two Real-time extensions for Java have been proposed. One is the "Experts Group Real-Time Specification for Java" (RTSJ) [RTJ00] and the other is the "J-Consortium Real-Time Core Extension" (RTCore) [JC00]. Implementations of these specifications are already available bringing the Java benefits to the real-time community.

In [CJWW02] the Distributed Real-Time Specification for Java (DRTSJ) introduces the Distributed Real-Time Remote Method Invocation (RMI) model. The DRTSJ is focused on supporting predictability of end-to-end timeliness for sequentially trans-node behaviours (e.g., chains of invocations) in dynamic distributed object systems. The integration and extension of the existing RTSJ and Java Remote Method Invocation (RMI) facility to provide the basis for the Distributed Real-Time Specification for Java (DRTSJ) is being investigated in [WCJW01] and [WCJW02].

The use of resource management strategies in supporting the testing and certification of real-time, fault-tolerant, mission critical systems based on distributed object middleware was explored in [WBV02].

In Chapter 5, it was presented a solution for the joint dispatching of tasks and messages using the FTT-CAN protocol, and it was also presented a nano-kernel for the station nodes.

When considering the benefits of a software architecture based on a JVM, this approach could be integrated in each station node and bring platform independence to FTT-CAN based systems. This integration was studied but its implementation was not accomplished on the works for this thesis. This study is presented in the following sections.

8.1 Architecture of a Distributed Embedded System

A typical distributed system is composed of several nodes interconnected by some network that can be wire-based or wireless. The architecture presented here uses a Java Virtual Machine (JVM) running on top of a basic kernel. This JVM allows the execution of special Java programs, called FTTlets [CF03b], which are executed upon the arrival of a triggering event from the FTT engine (See Figure 8-1).
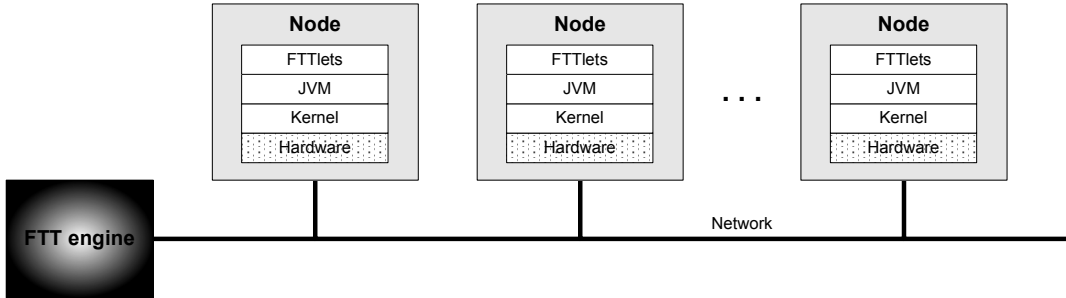
Figure 8-1 – A distributed embedded system

The FTT engine contains and manages FTTlets through their lifecycle. The FTT engine takes care of:

- FTTlet admission control;
- Scheduling;
- EC Trigger Message generation, this dispatches the FTTlets at each node and orders the transmission of messages;
- Kernel, JVM and FTTlet loading (not mandatory).

Currently this architecture can be implemented using the FTT-CAN protocol, [Alm99] and [APF02], or the FTT-Ethernet protocol [PAG02] to trigger the dispatching.

8.1.1 Kernel

The kernel offers services to the JVM in the form of an Application Programming Interface (API) and services to the FTT Engine in the form of a System Programming Interface

(SPI). These services mainly include inter-node task communications (message exchanging) and FTT engine event decoding.

The kernel is organized in several layers as depicted in Figure 8-2.

System Calls	
Dispatching	
List management	
Interrupt handling	Clock handling

Figure 8-2 – Kernel layers

The functions of the layers are:

- System Calls – services provided by the kernel to the FTT engine and the JVM;

- Dispatching – upon reception of an event from the FTT engine, this layer instructs the JVM about the next FTTlet to be dispatched;

- List management, interrupt handling and clock handling – basic kernel functionality.

The absence of a scheduler is justified by the centralized scheduling mechanism of the FTT.

The use of a kernel instead of an adapted JVM that handled directly the hardware, the network and the FTT implementation, has great benefits. For each architecture there has to be a specific implementation of the kernel but the JVM only needs to be recompiled.

8.1.2 Java Virtual Machine

The JVM is the platform of execution of the FTTlets and has to handle the intra-node FTTlet communication.

The API is made available in the form of packages. These packages include the standard Java classes (depending on the edition) and system specific packages that allow operations like Sending and Receiving messages (FTT package) and accessing the hardware features.

Due to the constant evolution of the JVMs, especially in what concerns real-time, the JVM is running on top of a kernel. This approach isolates the JVM from the underlying architecture. To accomplish this isolation the JVM has to be coupled with a set of native functions that interface it with the kernel.

The under-specified thread scheduler, an issue in the use of Java in real-time systems, is disabled. Just like in the kernel, the scheduling is handled by the centralized scheduling mechanism of the FTT.

The thread dispatching unit of the JVM is also disabled. So the dispatching is handled by the node kernel, upon reception of the trigger event.

8.1.3 FTTlet

Due to the use of the FTT paradigm, the FTTlets are Java programs (resembling servlets) that are started through the network, i.e. are executed upon arrival of a triggering event. This event is caught by the local kernel. The event contains information about the messages and tasks that should be dispatched. If there are messages to be dispatched the local kernel starts their transmission. If there are FTTlets selected for dispatching the local JVM is instructed to execute them. The FTTlets are loaded into each node, by the FTT engine, according to the requirements.

A FTTlet is a Java technology based component, managed by a FTT engine. Like other Java-based components, FTTlets are platform independent Java classes that are compiled to platform neutral bytecodes that can be loaded dynamically into any node and dispatched by a Java enabled FTT engine.

The key benefits of the FTTlets are:

- FTTlets are fast because they are loaded into memory once, and run from memory thereafter;
- FTTlets are relatively simple to implement;
- Since FTTlets are Java byte code, they are platform independent by nature.

8.2 System operation

8.2.1 FTT package

In order to offer the functions needed by the FTTlets to interoperate with the rest of the system a FTT package is available. This package provides a standard way to access the FTT features of the system.

The FTT package is based on the FTT class. This class offers the following methods:

- ReadMessage
- PostMessage
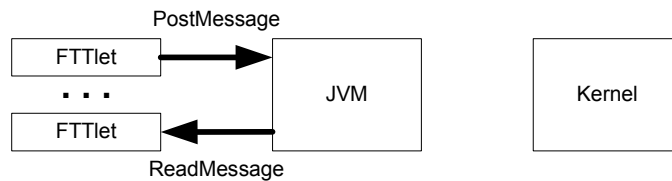
Figure 8-3 – Message transaction in the same node

If the message transaction is with a task running in the same JVM this package handles the transaction (see Figure 8-3). If the message transaction is with a different task then these methods interact with the kernel exchanging data between it and the FTTlets running in the JVM.

8.2.2 Kernel

The kernel is the cornerstone of the system because it allows the JVM and FTTlets to execute without being aware of the underlying system configuration. The kernel has to be adapted to each possible situation that depends on:

- The network topology,
- The transport protocol,
- If there is a real, or simulated, system.

The kernel offers two groups of services, namely: the application programming interface (API) and the system programming interface (SPI).

The API is a set of functions that are available to the JVM. These include:

- ReadMessage
- PostMessage

The method ReadMessage, is a blocking function, and is used to read a message sent by another task (or FTTlet).

The method PostMessage, is a non-blocking function, that is used to post a message to another FTTlet in a different node (see Figure 8-4). Therefore, the message interchange is the global communication paradigm that is used. One advantage of this model is the

possibility of changing the node of execution of a task without interfering with the other tasks.
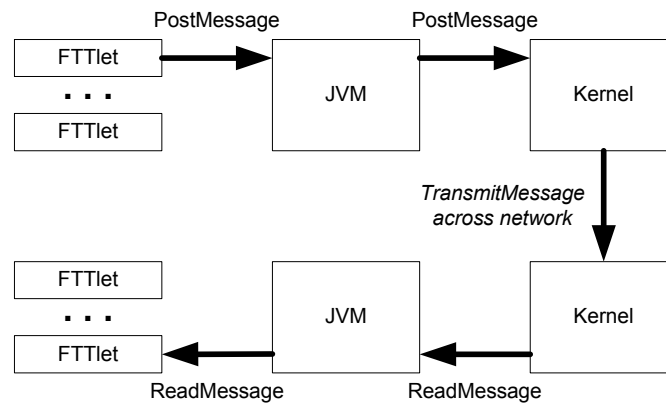
Figure 8-4 – Message transaction between different nodes

The SPI is a set of functions that are available to the FTT engine. These include:

- LoadJVM
- StartJVM
- LoadFTTlet
- StartFTTlet

The LoadJVM is used in the initialization procedure, where the FTT engine has to cooperate with the kernel for the loading of the JVM to the node's address space. The StartJVM is used to start the system. FTTlets can be loaded in a node through LoadFTTlet. The function StartFTTlet is basically an internal function that is called whenever the triggering event dispatches any FTTlet.

8.3 System simulation

One of the main benefits of this architecture is the possibility of developing and testing, transparently, the FTTlets in a simulated environment using a regular computer. This environment is supported by a special local kernel that is able to launch one JVM for each node of the simulation (see Figure 8-5). This local kernel simulates the triggering events and takes care of the communication. Each JVM is unaware of the environment bellow.
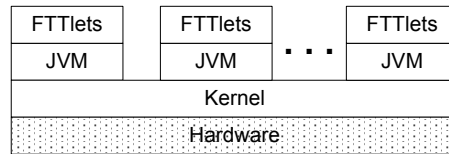
Figure 8-5 – Simulator environment

This approach facilitates the load balancing when node independent FTTlets are being executed.

Also, the system designer can make use of the simulation results in order to allocate features to each node, so that the nodes more heavily used can have some functionality moved to a different node.

8.4 Conclusions and future work

In this chapter, it was shown how to integrate a JVM with the joint dispatching of tasks and messages on a FTT-CAN based system. This JVM is a simple one because it does not require a scheduler or an actual dispatcher. This architecture permits the development of platform independent applications, named FTTlets. The FTTlets can be easily loaded, or migrated, to any station node bringing a significant flexibility to a system. This portability can also be explored in the simulation of a system. A simulator that handles the actual FTTlets can be developed for a particular hardware architecture.

A future work can be the implementation of this specific JVM and the corresponding simulator. Also the implementation of a framework that supports the development of FTTlets would also bring many benefits.

9 Conclusions and future work

The central proposition of this thesis, supported by the present dissertation, claims that the scheduling of real-time tasks and messages can benefit from approaches that result from a tighter coupling with parameter determination. The case-study adopted to validate this claim was the Flexible Time-Triggered CAN protocol: a protocol that combines centralized on-line scheduling with the support for time-triggered traffic in a flexible way.

The interactions between tasks were studied from the point of view of the producer/consumer model. These involve both unicast and multicast communication. From the task interaction scenarios of a given system, potentially, several data streams may be identified. A data stream shows an information flow between tasks. In more complex systems, the task interaction scenarios can result in interdependent data streams. Interdependence between data streams occur when a task participates in more than one data stream. The information flow of the process can be controlled offline or online. The online control combines the parameter setup before the system starts to function with an adaptive run-time system adjustment. This parameter determination and tuning takes place whenever there is any change in the tasks or messages in the system.

The data stream analysis is beneficial in different phases of a distributed system planning. This analysis is useful in situations like: computational and network load evaluation, early stage system parameters definition, real-time procedures, when the various entities don't have specific deadlines; systems with a high network load, where the messages have strict deadlines; systems with a high node load, where the tasks have strict deadlines.

At an early stage where very few parameters are defined, apart from the execution times and message size, this analysis helps in the definition of suitable values for the initial phases and the deadlines. After this, the system developer can fine tune any parameter according to some special needs. If, on the contrary, almost every parameter is clearly defined, then this analysis is useful to check the assumptions.

A procedure is a conceptual set of operations that might be implemented through a set of tasks, where each operation does not necessarily map to each task. A real-time procedure has, at least, one timing constraint. Usual constraints are the period and the deadline, but others can also be defined. For the case where a procedure is mapped to more than one

task, an approach has to be chosen in order to define the timing parameters of each, correspondent, task and message. Simple algorithms were proposed in order to accomplish the mapping of parameters for the three main approaches previously presented.

Naturally, this flexibility comes at the cost of an additional complexity. It was shown that this complexity can be bounded because all iterative algorithms have a limited number of iterations. It can also be implemented within an acceptable cost frame because its implementation can make use of the computational resources of the node where the admission control takes place.

Each task has its own characteristics, namely: execution time, running node and type of interaction with other tasks. So, in order to achieve a feasible schedule for the messages, tasks have also to be considered in a holistic scheduling. In a system with several tasks running in several nodes, a node may have more than one task assigned.

In order to make use of the FTT-CAN protocol to accomplish a centralized dispatching of both tasks and messages, a fundamental extension to it was proposed. This extension was the joint dispatching of tasks and messages. To accomplish this goal, the EC trigger message was redesigned to accommodate also in its data field, apart from the synchronous messages that must be transmitted and additional coding, an indication of which tasks must be started in the current EC. This way, the EC trigger message can be used to also trigger remote task execution. It was shown that with this simple extension, the overhead introduced was small because each task was assigned a single bit in a trigger message.

The kernels of the master node and the station nodes were also addressed and a simple, but effective, nano-kernel was proposed. This simple kernel introduces a minimal overhead because it does not have any scheduling unit. The case study of the CAMBADA robots demonstrated that such lightweight kernels can be used even with low processing power nodes. All the complexity of the parameter setup, scheduling and dispatching can be accomplished by a master node. So for this node, a more elaborate kernel is required. Also, a technique based on the data streams is proposed in order to determine the level of buffering, in each time slot, for each kernel of the station nodes. With this, a system designer can more easily define the memory requirements for the kernel of each node.

A simulator, SimHol, which supports the joint scheduling and dispatching of messages and tasks, was proposed. The SimHol offers a suitable experimental platform for the simulation of distributed systems based on the time-triggered (FTT) paradigm. Using a simple interface it allows the simulation of various interconnection architectures, although currently only CAN is supported. Different scheduling algorithms can also be chosen. The scheduling takes place at an EC basis closely resembling an actual system. This simulator has validated the set of requirements previously derived, namely the data flow analysis and the precedence requirements.

The thesis stated in Chapter 1, arguing that the scheduling of real-time tasks and messages can benefit from approaches that result from a tighter coupling with parameter determination was supported throughout this dissertation for the specific case of FTT-CAN. In fact, it has been shown, mainly with the work presented in Chapters 4, 5, 6 and 7, that it is possible to improve the scheduling from different perspectives by following the proposed approaches.

9.1 Future research

Formal validation of the data stream analysis and the real-time procedures

The use of a simple form of representation for the interactions between tasks in the development of the proposed approaches permitted focusing on the aspects considered more relevant for this thesis. After this work, it becomes simpler to evaluate other methodologies for the design of real-time systems, like the ones presented in Chapter 4, and distinguish clearly the ones that are able to meet the same goals of the data streams while providing mechanisms for a formal validation. These methodologies must also be able to provide the abstraction of the real-time procedures while not being overly resource demanding, both in processing and in memory requirements.

Optimization of the transmission and execution windows

After this work, some optimizations are foreseeable, namely some work concerning the optimization of the transmission and execution windows, according to the approach followed, can be developed in the sequence of this one. In fact, currently, any remaining time after satisfying the execution and transmission windows can be used to relax some of

these windows. To accomplish this, other criteria might be necessary to complement the ones used in the approaches.

Methodologies for data stream shifting

Currently, all data streams are assigned the same initial phase. This policy is not based in any reasoning and it constitutes an area that can be addressed. Thus, a future work is the research in methodologies for data stream shifting so that a criterion like the jitter can be minimized.

Improve the decoupling between the transmission and execution windows through buffering

It was shown how to decouple the transmission and execution windows of consecutive entities in a data stream through the use of buffering. This work can be extended by introducing this extra flexibility into the parameter determination and tuning.

Using execution time measures for improving the parameters of tasks

With appropriate support form the hardware, the execution time of tasks can easily be measured. If this parameter can be made available to the software layers then the parameters of tasks can be improved. The integration of this measure directly into a processor is being studied in [OSF05]. The Advanced Real-time Processor Architecture (ARPA) includes a processor with a deterministic performance. According to the authors, this processor can easily include mechanisms for measuring the execution time of tasks. A future work can be the development of methodologies for the online use of these measures in a system with a processor based on the Advanced Real-time Processor Architecture (ARPA). This system would evolve with time through the tuning of the tasks' and messages' parameters.

Complete the implementation of the SimHol and improve the XML formats for input/output data

Although the design of a version of the SimHol that implements all the approaches was done, its implementation was only partially accomplished. Due to its modularity, which facilitates a gradual development, the implementation of the SimHol should be completed.

The possibility of introducing online changes is also very important to broaden the simulation scenarios. Also, the XML formats defined as input and output of the modules of the SimHol can be further improved so that eventually they can promote the interaction between tools based on the FTT-CAN.

Implement all the features of the station kernel of the CAMBADA robots

The kernel of the station nodes of the CAMBADA robots is only partially functional. The message handling is already accomplished but the same cannot be said about the task handling and the application programming interface. Therefore, as future work, the implementation of all this functionalities is foreseeable.

Bibliography

[AF00]   Almeida L., J. Fonseca, "FTT-CAN: A Network-Centric Approach for
             CAN-based Distributed Systems", 4th IFAC Symposium on Intelligent
             Components and Instruments for Control Applications (SICICA'00),
             Buenos Aires, Argentina, September 2000.

[Alm99]   Almeida, L., "Flexibility and Timeliness in Fieldbus-based Real-Time
             Systems", PhD thesis, University of Aveiro, Aveiro, Portugal, 1999.

[AFF98]   Almeida, L., J.A. Fonseca, P. Fonseca, "Flexible time-triggered
             communication on a controller area network", Proceedings of the Work-In-
             Progress Session of the 19th IEEE Real-Time Systems Symposium
             (RTSS'98), Madrid, Spain, 1998.

[AHC05]   Andersson, M., D. Henriksson and A. Cervin, "TrueTime 1.3—Reference
             Manual", Department of Automatic Control, Lund University, Sweden, June
             2005.

[APF99]   Almeida, L., R. Pasadas, J.A. Fonseca, "Using a planning scheduler to
             improve flexibility in real-time fieldbus networks", Control Engineering
             Practice, 7:101-108, 1999.

[APF02]   Almeida, L. , P. Pedreiras, J.A. Fonseca, "The FTT-CAN protocol: Why and
             How", IEEE Transactions on Industrial Electronics – special edition on
             Factory Communication Systems, Volume 49, Issue 6, pp. 1189-1201,
             December 2002.

[ASF+04]   Almeida, L., Santos, F., Facchinetti, T., Pedreiras, P., Silva, V., Lopes, L.,
             "Coordinating distributed autonomous agents with a real-time database: The
             CAMBADA project", Proceedings of the 19th Int. Symp. on Computer and
             Information Sciences (ISCIS 2004), Kemer-Antalya, Turkey, October 27-
             29, 2004.

[BBGT96]   P. Bizzarri, A. Bondavalli, F. Giandomenico, F. Tarini, "Planning the
             Execution of Task Groups in Real-Time Systems", Proceedings of the 8th

IEEE Euromicro Workshop on Real-Time Systems (WRTS'96), L'Aquila, Italy, June 1996.

[BCR⁺99]   Beccari, G., C. Caselli, M. Reggiani, F. Zanichelli, "Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems", Proceedings of the 11th Euromicro Conference on Real-Time Systems (RTS'99), York, UK, June 1999.

[Boo93]   Booch, G., "Software Engineering with Ada", 3rd edition, Addison-Wesley Professional, 1993.

[Bor02]   Borland Software Corporation, "C++ Builder 5", http://www.borland.com/es/products/cbuilder/index.html, 2002.

[Bos91]   Bosch, Robert [1991], CAN Specifications Version 2.0, BOSCH, Stuttgart, Germany.

[BS05]   Bouyssounouse, B., J. Sifakis, "Embedded Systems Design", Series: Lecture Notes in Computer Science, Vol. 3436, Springer, 2005.

[But00]   Buttazzo, G., "Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications", Kluwer Academic Publishers, 2000.

[Can96]   CiA DS 201, "CAN Application Layer for Industrial Applications", CiA, CAN in Automation International Users and Manufacturers Group, 1996.

[Car05]   Carlson, D., "Eclipse distilled", Addison-Wesley Publishing Company Inc, 2005.

[CDK05]   Coulouris, G., J. Dollimore, T. Kindberg, "Distributed Systems – Concepts and Design", 4th edition, Addison-Wesley, 2005.

[Cer99]   Cervin, A., "Improving scheduling of control tasks". Proceedings of the 11th Euromicro Conference on Real-Time Systems (RTS'99), York, UK, June 1999.

[CF02]   Calha, M.J., J.A. Fonseca, "Adapting FTT-CAN for the joint dispatching of tasks and messages", Proceedings of the 4th IEEE International Workshop

on Factory Communication Systems (WFCS'02), Vesteras, Sweden, August 2002.

[CF03a]   Calha, M.J., J.A. Fonseca, "SIMHOL – A graphical simulator for the joint scheduling of messages and tasks in distributed embedded systems", 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2003), Aveiro, Portugal, July 2003.

[CF03b]   Calha, M.J., J.A. Fonseca, "FTTlet based distributed system architecture", 2nd International Workshop on Real-Time LANs in the Internet Age (RTLIA2003), Porto, Portugal, July 2003.

[CF04]   Calha, M.J., J.A. Fonseca, "Approaches to the FTT-based scheduling of tasks and messages", Proceedings of the 5th IEEE International Workshop on Factory Communication Systems (WFCS'04), Vienna, Austria, September 2004.

[CF05]   Calha, M.J., J.A. Fonseca, "Data Streams – an Analysis of the Interactions Between Real-Time Tasks", Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005), Catania, Italy, September 2005.

[CJWW02]   Clark, R., Jensen, E., Wellings, A. and Weels, D., "The Distributed Real-Time Specification for Java: A Status Report", Embedded systems conference, San Francisco, USA, March 2002.

[CP00]   P. Chevochot, I. Puaut, "Holistic Schedulability Analysis of a Fault-Tolerant Real-Time Distributed Run-time Support", 7th IEEE International Conference on RealTime Computing Systems and Applications (RTCSA'00), Cheju Island, South Korea, December 2000.

[CSF05]   Calha, M.J., V.F. Silva, J.A. Fonseca, "Real-time procedures in distributed systems", Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2005), Puebla, Mexico, November 2005.

[CSFM06]   Calha, M.J., V.F. Silva, J.A. Fonseca, R. Marau, "Kernel design for FTT-CAN systems", To appear in the Proceedings of the 6th IEEE International

Workshop on Factory Communication Systems (WFCS'06), Torino, Italy, June 2006.

[CSM97]   Cavalieri, S., Stefano, A., Mirabella, O., "Impact of Fieldbus on Communication in Robotic Systems", IEEE Transactions on Robotics and Automation, Vol. 13, N. 1, February 1997.

[DHT01]   Dashofy, E. M., A. Hoek, R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language", Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), Amsterdam, Netherlands, August 2001.

[Dou04]   Douglass, B. P., "Real time UML – advances in the UML for real-time systems", Addison-Wesley Publishing Company Inc, 2004.

[FAM$^+$03]   J. Ferreira, L. Almeida, E. Martins, P.Pedreiras, J.A. Fonseca, "Enforcing Consistency of Communication Requirements Updates in FTT-CAN", Workshop on Dependable Embedded Systems, SRDS2003, 22nd Symposium on Reliable Distributed Systems, Florence, Italy, 6-8 October, 2003.

[Fer05]   Ferreira, J., "Fault-Tolerance in Flexible Real-Time Communication Systems", PhD thesis, University of Aveiro, Aveiro, Portugal, 2005.

[FFC$^+$02]   Fonseca, J.A., J. Ferreira, M. Calha, P. Pedreiras, L. Almeida (2002). "Issues on Task Dispatching and Master Replication in FTT-CAN", Africon'2002 – IEEE International Conference, George, South Africa, October 2002.

[FMD$^+$00]   Führer, T., Müller, B., Dieterle, W., Hartwich, F., Hugel, R., Walther, M., Bosch, R., "Time Triggered Communication on CAN (Time-triggered CAN-TTCAN)", Procedings of ICC'2000, Amsterdam, Netherlands, 2000.

[Gom84]   Gomaa, H., "A Software Design Method for Real Time Systems", CACM, vol. 27, no. 9, pp. 938-949, September 1984.

[Har87]   Harel, D., "Statecharts: A Visual Formalism for Complex Systems", Sci. Comput. Programming 8, 231-274, 1987.

[Har88]   Harel, D., "On Visual Formalisms", *Comm. Assoc. Comput. Mach.* 31:5, 514-530, 1988.

[HC05]   Horstmann, C. S., G. Cornell, "Core Java 2", Vols. 1 & 2, 7th edition, Sun Microsystems Press, 2005.

[HCA03]   Henriksson, D., A. Cervin and K.E. Årzén, "TrueTime: Real-time Control System Simulation with MATLAB/Simulink", Proceedings of the Nordic MATLAB Conference, Copenhagen, Denmark, October 2003.

[HKRB98] Henderson, W.D., D. Kendall, A.P. Robson, and S.P. Bradley, "Xrma: An holistic approach to performance prediction of distributed real-time CAN systems", Proceedings of Can In Automation Conference, San Jose, 1998.

[Ibm03]   IBM Corporation. "Eclipse platform", Technical report, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, February, 2003.

[Int93]   International Organization for Standardization ISO 11898-2, Road Vehicles – Controller Area Network (CAN) – Part 2: High-speed medium access unit, 1993.

[Int94]   International Organization for Standardization ISO 11519-2, Road Vehicles – Controller Area Network (CAN) – Part 2: Low-speed serial data communication, 1994.

[Int00]   International Organization for Standardization ISO 11898-4, Road Vehicles – Controller Area Network (CAN) – Part 4: Time-Triggered Communication, 2000.

[Jac83]   Jackson, M., "System Development", Prentice Hall PTR, 1983.

[JC00]   J Consortium, "Real-Time Core Extensions", 2000.

[KA02]   Kongezos, V., Allen, C.R., "Wireless Communication between A.G.V.'s (Autonomous Guided Vehicle) and the industrial network C.A.N. (Controller Area Network)", Proc. 2002 IEEE Int. Conf. on Robotics & Automation Washington, DC, May 2002.

[Koo02]   Koopman, P., "Critical Embedded Automotive Networks", IEEE Micro, IEEE Press, July/August 2002.

[Kop97]   Kopetz, H., "Real-Time Systems: Design Principles for Distributed Embedded Applications", Kluwer Academic Publishers, 1997.

[Kop99a]   Kopetz, H., "Specification of the TTP/C Protocol, version 0.5", Technical report, Document edition 1.0, TTTech Computertechnik AG, July 1999.

[Lap97]   Laplante, P.A., "Real-time systems design and analysis – an engineer's handbook", IEEE Press, 2nd edition, 1997.

[LL73]   Liu, C. L., and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM 20(1), 46-61, 1973.

[LW82]   Leung, J., Whitehead, J., "On the complexity of fixed-priority scheduling of periodic, real-time tasks", Performance evaluation 2, 237-250, 1982.

[MD78]   Mok, A., Detouzos, M., "Multiprocessor scheduling in a hard real-time environment", Proceedings of the 7th IEEE Texas Conference on Computing Systems, November 1978.

[MF01]   Martins, E., J. Fonseca, "Improving flexibility and responsiveness in FTT-CAN with a scheduling coprocessor", Proceedings of the 4th IFAC Conference on Fieldbus Technology (FET'01), Nancy, France, November 2001.

[MFA+02]   Martins, E., J. Ferreira, L. Almeida, P. Pedreiras, J.A. Fonseca. An Approach to the Synchronization of Backup Master in Dynamic Master-Slave Systems. Work-in-Progress Session of RTSS 2002, IEEE 23rd Real-Time Systems Symposium, Austin, USA, December 2002.

[MNF02]   Martins, E. , P. Neves, J. A. Fonseca, "Architecture of a Fieldbus Message Scheduler Coprocessor Based on the Planning Paradigm", Microprocessors and Microsystems, Vol. 26, Issue 3, April 2002.

[MN99]   Mock, M., Nett, E., "Real-Time Communication in Autonomous Robot Systems", Proceedings of the 4th International Symposium on Autonomous Decentralized Systems, 1999, Integration of Heterogeneous Systems, pp. 34-41, 21-23 March 1999.

[MT00]   Medidovic, N., R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering. 26(1):70-93. January 2000.

[NHNP01]   Nolte, T., Hansson, H., Norstrom, C., Punnekkat, S., "Using bit-stuffing distributions in CAN analysis", IEEE Real-time Embedded Systems Workshop, December 2001.

[OSF05]   Oliveira, A.S.R., V.A. Sklyarov, A.B. Ferrari, "ARPA - A Technology Independent and Synthesizable System-on-Chip Model for Real-Time Applications", actas da conferência DSD 2005 - Euromicro Conference on Digital System Design, Porto, Portugal, Agosto 2005, pp. 484-491.

[PAG02]   Pedreiras, P., L. Almeida and P. Gai, "The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency", 14th Euromicro Conference on Real-Time Systems, Viena, Austria, June 2002.

[PBB+03]   Pérez, P., G. Benet, F. Blanes, J.E. Simó, Communication Jitter Influence on Control Loops Using Protocols for Distributed Real-Time Systems on CAN bus, Proceedings of IFAC SICICA 2003, Aveiro, Portugal, July 2003.

[Ped03]   Pedreiras, P., "Supporting Flexible Real-Time Communication on Distributed Systems", PhD thesis, University of Aveiro, Aveiro, Portugal, 2003.

[PEP02]   Pop, T., P. Eles, Z. Peng, "Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems", 10th International Workshop on Hardware/Software Codesign (CODES 2002), Estes Park, Colorado, USA, May 2002.

[PEP03]   Pop, T., P. Eles, Z. Peng, "Schedulability Analysis for Distributed Heterogeneous Time/Event Triggered Real-Time Systems",15th Euromicro Conference on Real-Time Systems (ECRTS 2003), Porto, Portugal, July 2003.

[Pet62]   Petri, C.A., "Kommunikation mit Automaten", PhD thesis, Faculty of Mathematics and Physics at the Technische Universität Darmstadt, Germany, 1962.

[PH98]   Palencia, J.C., M.G. Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets", 19th IEEE Real-Time Systems Symposium (RTSS'98), Madrid, Spain, December 1998.

[PH99]   Palencia, J.C., M.G. Harbour, "Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems", 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, USA, November 1999.

[Pic05]   Microchip website, "PICmicro Microcontrollers" available at the Literature section of www.microchip.com, 2005.

[Pre92]   Pressman, R., "Software Engineering – A Practitioner's Approach", McGraw-Hill Inc, 1992.

[Ram95]   Ramamritham, K., "Allocation and Scheduling of Precedence-Related Periodic Tasks", IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 4, April, 1995.

[RCR01]   Richard, P., F. Cottet, M. Richard, "On-line Scheduling of Real-Time Distributed Computers With Complex Communication Constraints", 7th International Conference on Engineering of Complex Computer Systems (ICECCS'01), Skovde, Sweden, June 2001.

[RJB98]   Rumbaugh, J., I. Jacobson, and G. Booch, "The Unified Moddeling Language Reference Manual", Addison-Wesley Publishing Company Inc, 1998.

[Rob91]   Bosch, R., "CAN Specification version 2.0", Stuttgart, Germany, 1991.

[RRC01]   Richard, M., P. Richard, F. Cottet, "Task and Message Priority Assignment in Automotive Systems", 4th IFAC Conference on Fieldbus Technology (FET'01), Nancy, France, November 2001.

[RRC03]   Richard, M., P. Richard, F. Cottet, "Allocating and Scheduling Tasks in Multiple Fieldbus Real-Time Systems", 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003), Lisbon, Portugal, September 2003.

[RTJ00]   The Real-Time for Java Expert Group, "The Real-Time Specification for Java", Addison-Wesley Publishing Company Inc, 2000.

[Rus01]   Rushby, J., "Bus Architectures For Safety-Critical Embedded Systems", Proceedings of the First Workshop on Embedded Software, Lecture Notes in Computer Science vol. 2211, pp 306-323, 2001.

[SAA+04]   Sha, L., T. Abdelzaher, K. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M.. Caccamo, J. Lehoczky, A. Mok, "Real- Time Scheduling Theory: A Historical Perspective", The International Journal of Time-Critical Computing Systems, Vol. 28, Num. 2/3, November/December, 2004.

[SFN+05]   Silva, V., J.A. Fonseca, U. Nunes, R. Maia, "Communications requirements for autonomous mobile robots: analysis and examples", Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2005), Puebla, Mexico, November 2005.

[SGG04]   Silberschatz, A., P.B. Galvin, G. Gagne, "Operating system concepts", John Wiley & Sons, 7th edition, December, 2004.

[Sch99]   Schultz, T.W., "C and the 8051", vols. 1 & 2, Prentice Hall PTR, 1999.

[Sim04]   Simmons, R., "Hardcore Java", O'Reilly Media Inc., 2004.

[SMA+05]   Silva, V., R. Marau, L. Almeida, J. Ferreira, M. Calha, P. Pedreiras, J. Fonseca, "Implementing a distributed sensing and actuation system: The CAMBADA robots case study", Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005), Catania, Italy, September 2005.

[Som95]   Sommerville, I., "Software Engineering", Addison-Wesley Publishing Company Inc, 1995.

[Spe00]   Spencer, J., ed. Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools. Open Group White Paper. September 26, 2000.

[SSL89]   Sprunt, B., L. Sha, and J. Lehoczky. "Aperiodic task scheduling for hard
            real-time systems", *Journal of Real-Time Systems*, 1(1):27-60, 1989

[Ste84]   Steusloff, H. U., "Advanced Real-Time Languages for Distributed Industrial
            Process Control", IEEE Computer Mag., vol. 17, no. 2, pp. 37-46, February
            1984.

[Str97]   Stroustrup, B., "The C++ programming language", Addison-Wesley
            Publishing Company Inc, 3rd edition, 1997.

[Sun05]   Sun Microsystems Inc. "J2SE 5.0 documentation",
            http://java.sun.com/docs/index.html, 2005.

[TC94]   Tindell, K., J. Clark, "Holistic Schedulability Analysis for Distributed Hard
            Real-Time Systems", Microprocessing & Microprogramming 40, 117-134,
            1994.

[Tho98]   Thomesse, J.P., "A review of the fieldbuses", Annual reviews in control, 22
            pp.35-45, 1998.

[Vec02]   Vector Informatik GmbH (2002). "CANoe/DENoe", Product information at
            http://www.vector-informatik.de

[Ves93]   Vestal, S., "A Cursory Overview and Comparison of Four Architecture
            Description Languages", Technical Report, Honeywell Technology Center.
            February 1993.

[Vid83]   Vidondo, F., "GALILEO: Design Language for Real-Time Systems", Proc.
            ITT Conf. on Programming Productivity and Quality, ITT Corporation, pp.
            198-210, June 1983.

[VR01]   Veríssimo, P., Rodrigues, L., "Distributed Systems for System Architects",
            Kluwer Academic Publishers, 2001.

[VSC+99]   Valera, A., Salt, J., Casanova, V., Ferrus, S., "Control of Industrial Robot
            With a Fieldbus", Proceedings of the 7th IEEE International Conference on
            Emerging Technologies and Factory Automation. ETFA '99. Vol. 2, pp. 18-
            21, Barcelona, Spain, October 1999.

[WBV02]   Wells, D., Bernstein, R. and Vadlamudi, A., "Testability of Complex, Middleware-Based Systems", Workshop on Dependable Middleware-Based Systems, Bethesda, Maryland, USA, June 2002.

[WCJW01]   Wellings, A., Clark, R., Jensen, D. and Wells, D., "Towards a Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", The 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, December 2001

[WCJW02]   Wellings, A., Clark, R., Jensen, D. and Wells, D., "A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation", Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C., USA, April 2002.

[Wit85]   Witt, B. I., "Communication Modules: A Software Design Model for Concurrent Distributed Systems", IEEE Computer, vol. 18, no. 1, pp. 67-77, January 1985.

[WM86]   Ward, P. T., and S. J. Mellor, "Structured Development for Real-Time Systems", 3 volumes, Prentice-Hall/Yourdon Press, 1985, 1986.

[XP90]   Xu, J., D.L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations", IEEE Transactions on Software Engineering, vol.16, no.3, p.360-369, March 1990.

[XP91]   Xu, J., D.L. Parnas, "On satisfying timing constraints in hard-real-time systems", ACM SIGSOFT Software Engineering Notes, v.16 n.5, p.132-146, December 1991.

[YBP+04]   Yergeau, F., T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler. "Extensible Markup Language (XML) 1.0", W3C Recommendation, 3rd edition, http://www.w3.org/TR/2004/REC-xml-20040204, February, 2004.

[YCB+04]   Yergeau, F., J. Cowan, T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler. "Extensible Markup Language (XML) 1.1", W3C Recommendation, http://www.w3.org/TR/2004/REC-xml11-20040204, February, 2004.

[YPS+02]   Yagüe, J. L. P., P. Pérez, J. Simó, G. Benet and F. Blanes. Communications structure for sensory data in mobile robots. Engineering Applications of Artificial Intelligence 15, pp341-350, 2002.

Appendix A

List of publications and communications

In the scope of the research developed towards the preparation of this PhD thesis, the following documents have been published:

A.1 Conference papers

Papers with the candidate as first author:

[CF02]   Calha, M.J., J.A. Fonseca, "Adapting FTT-CAN for the joint dispatching of tasks and messages", Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02), Vesteras, Sweden, August 2002.

[CF03a]   Calha, M.J., J.A. Fonseca, "SIMHOL – A graphical simulator for the joint scheduling of messages and tasks in distributed embedded systems", 5th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2003), Aveiro, Portugal, July 2003.

[CF03b]   Calha, M.J., J.A. Fonseca, "FTTlet based distributed system architecture", 2nd International Workshop on Real-Time LANs in the Internet Age (RTLIA2003), Porto, Portugal, July 2003.

[CF04]   Calha, M.J., J.A. Fonseca, "Approaches to the FTT-based scheduling of tasks and messages", Proceedings of the 5th IEEE International Workshop on Factory Communication Systems (WFCS'04), Vienna, Austria, Sep/2004.

[CF05]   Calha, M.J., J.A. Fonseca, "Data Streams – an Analysis of the Interactions Between Real-Time Tasks", Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005), Catania, Italy, September 2005.

[CSF05]   Calha, M.J., V.F. Silva, J.A. Fonseca, "Real-time procedures in distributed systems", To appear in the Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2005), Puebla, Mexico, November 2005.

[CSFM06]   Calha, M.J., V.F. Silva, J.A. Fonseca, R. Marau, "Kernel design for FTT-CAN systems", To appear in the Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS'06), Torino, Italy, June 2006.

Other papers co-authored by the candidate:

[FFC⁺02]   Fonseca, J.A., J. Ferreira, M. Calha, P. Pedreiras and L. Almeida (2002). "Issues on Task Dispatching and Master Replication in FTT-CAN", Africon'2002 – IEEE International Conference, George, South Africa, October 2002.

[SMA⁺05]   Silva, V., R. Marau, L. Almeida, J. Ferreira, M. Calha, P. Pedreiras, J. Fonseca, "Implementing a distributed sensing and actuation system: The CAMBADA robots case study", Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005), Catania, Italy, September 2005.