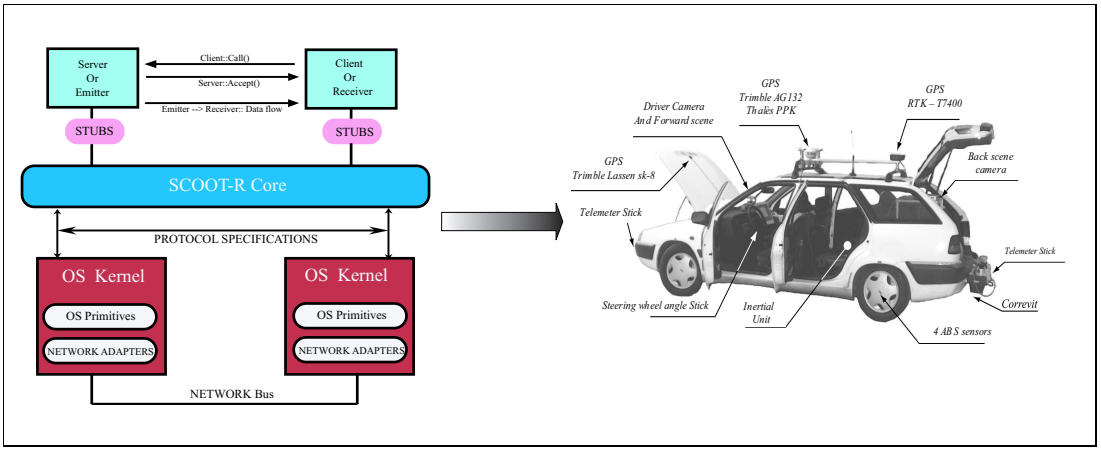
# A Distributed Real-Time Architecture For Advanced Vehicles

Khaled Chaaban

By Khaled Chaaban

# *A Distributed Real-Time Architecture For Advanced Vehicles*

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Computer Science
Université de Technologie de Compiègne (UTC).




Defended: June 2006
Option: Technologies de l'information et des systèmes

# Architecture Informatique Temps-Réel Pour Véhicules Avancés

By Khaled Chaaban

Thesis defended on June 2006

Doctoral committee:

| | | | |
|---|---|---|---|
| Mme. | Isabelle Puaut | Professeur, IRISA, Université de Rennes | (rapporteur) |
| Mr. | Yvon Trinquet | Professeur, IRCCyN, Nantes | (rapporteur) |
| Mr. | Laurent George | Mdc, ECE, Paris | (examinateur) |
| Mr. | Guy Friedrich | Professeur, LEC, UTC | (examinateur) |
| Mr. | Mohamed Shawky | Mdc (HDR), Heudiasyc, UTC | (directeur) |
| Mr. | Paul Crubillé | Ingénieur de recherches CNRS, Heudiasyc, UTC | (co-directeur) |
| Mr. | Pavol Barger | Mdc, UTC | (Invité) |

*For my parents and my dear wife Rouba*

# *Abstract*

During the recent years, car manufacturers have been developing new Advanced Driving Assistance Systems (ADAS) such as Collision Warning, Lane Keeping, Night Vision and Navigation systems, etc. To assess their real impacts on improving the driver behavior, car manufactures need adequate and efficient tools. A set of metrics has been pointed out that truly reflects the driver performances like Driver Reaction Time, Time To Collision, etc. To compute such metrics, an experimental car is equipped with numerous types of sensors, whose data is acquired and processed in real-time. All these requirements have lead us to the development of an object oriented middleware called SCOOT-R[1].

SCOOT-R consists of a set of basic services built as a middleware layer above an existing real-time kernel. It offers a framework for distributing tasks on multi-processing units architecture, along with communication and synchronization services. It also includes run-time monitoring of real-time constraints and ensures a dynamic reconfiguration by replicated software components. SCOOT-R aims to reduce the cost and time of distributed real-time applications development by providing the framework necessary for building reusable multi-purpose real-time software components. The benefits of our middleware are demonstrated by building a testbed and executing representative applications such as driving assistance applications.

Moreover, middleware must support applications with real-time QoS requirements. Our middleware SCOOT-R provides an ideal platform to design and implement distributed scheduling strategies that permit to ensure end-to-end real-time QoS capabilities. Thus, we have enhanced SCOOT-R client/server model by incorporating end-to-end scheduling strategies. The developed scheduling techniques consist of timing constraints propagation of remote operations. Three strategies are proposed in this context, the client priority propagation, the client deadline propagation (EDF), and finally a *hybrid* strategy is proposed to take into account the criticalness constraints with the distributed EDF scheduling strategy.

Finally, we have developed adaptive scheduling strategies to schedule at run-time driving assistance functions in presence of driving situation change. The adaptation in our approach is carried out on the *driving situation*, which will lead to the change of the associated driving assistance function's *criticalness*.

[1]SCOOT-R: Serveur et Client Orienté Objet pour le Temps-Réel

# *Résumé*

Ces dernières années, de nouveaux systèmes d'aide à la conduite, comme les systèmes de prévention de collision, ou de sortie de route, d'aide à la vision de nuit et à la navigation, etc., ont été développés par les constructeurs automobiles. Afin d'évaluer l'impact de ces systèmes d'aide sur l'amélioration de la sécurité et le comportement du conducteur, les constructeurs automobiles ont besoin d'outils assez flexibles et efficaces. Un ensemble de métriques a été défini afin d'évaluer les performances du conducteur face au système d'aide et de la situation de conduite à laquelle il fait face. Afin de calculer une telle métrique, un véhicule expérimental est équipé de nombreux types de capteurs, dont les données doivent être acquises et traitées en temps-réel. Toutes ces conditions ont motivé le développement d'une architecture distribuée appelée SCOOT-R.

SCOOT-R est un intergiciel orienté objet qui permet l'échange d'objets entre des processus s'exécutant sur des calculateurs reliés par un réseau selon un modèle client/serveur ou émetteur/récepteur en respectant des échéances temporelles. Il offre aux applications des services de *synchronisation* et de contrôle des contraintes temporelles, ainsi que la *configuration dynamique* par la réplication des composants logicielles. SCOOT-R vise à réduire le coût et le temps du développement des applications distribuées temps-réel en fournissant le cadre nécessaire pour le développement des composants logiciels temps-réel réutilisables. Les avantages de SCOOT-R sont démontrés en construisant un banc d'essai et en développant des applications représentatives telles que les fonctions d'aide à la conduite.

Un intergiciel temps-réel doit supporter des applications temps-réel avec des contraintes temporelles plus ou moins strictes. SCOOT-R représente une plateforme idéale pour la conception et le développement des stratégies d'ordonnancement distribués afin de garantir des communications de bout-en-bout prévisibles. Nous avons amélioré le modèle client/serveur de SCOOT-R en incorporant des stratégies d'ordonnancement bout-en-bout. Ces techniques d'ordonnancement développées consistent essentiellement en la propagation des contraintes temporelles. Trois stratégies ont été proposées dans ce contexte, la propagation de la priorité des clients, la propagation de l'échéance des clients (EDF) et finalement une stratégie hybride est proposée pour tenir compte les contraintes de criticité des opérations avec la stratégie d'ordonnancement distribuée EDF.

Finalement, nous avons développé des stratégies d'ordonnancement adaptatif (feedback scheduling) pour ordonnancer des fonctions d'aide à la conduite en présence de changement de situation de conduite. L'adaptation dans notre approche est porté sur la situation de conduite, qui mènera au changement de la criticité des fonctions d'aide à la conduite associées.

# *Acknowledgements*

This thesis has been conducted at *Heudiasyc* laboratory, among the *advanced vehicle* team and with the support of the European project RoadSense (ROad Awareness for Driving via a Strategy that Evaluates Numerous Systems).

This thesis would not have been possible if not for the help and encouragement from teachers, friends and loved ones.

I am in great debt to my mentors and supervisors, Mohamed Shawky and Paul Crubillé. They were an example to follow and from them I have learned so much. The amount of time and effort they have spent with me on never ending drafts of the thesis, and their wise and precise comments have contributed significantly to the quality of this thesis.

I would like to thank also my reviewers, Isabelle Puaut and Yvon Trinquet for their relevant and precise comments concerning the thesis dissertation and that lead to a high quality document.

I also want to thank all the members of PACPUS team at our laboratory. Specially, Ali Charara, Philippe Bonnifait, Véronique Cherfaoui and Dominique Meizel.

I have always received encouragement and support from my close friends, specially Amadou, Fahed, Géry, Gérarld and Olivier. I will be always in debt with them.

I am thankful to all the members of the Heudiasyc laboratory at the UTC university for their support and good moments.

Compiègne, France 27th January 2007

# *Publications*

## Papers

[CCS04] K. Chaaban, P. Crubillé, and M. Shawky. *Real-Time Framework for Distributed Embedded Systems*, In Principles of Distributed Systems, volume 3144 of Lecture Notes in Computer Science, pages 96 - 107. Springer-Verlag GmbH, January 2004.

## International conferences

[CSC05] K. Chaaban, M. Shawky, and P. Crubillé. *A distributed framework for real-time in vehicle applications*. In proceedings of the IEEE Conference on Intelligent Transportation Systems ITSC, Vienna, Austria, September 2005.

[CSC04] K. Chaaban, M. Shawky, and P. Crubillé. *Dynamic reconfiguration for high level in-vehicle applications using IEEE-1394*. In proceedings of the IEEE Conference on Intelligent Transportation Systems (ITSC), Washington, D.C, October 2004.

[CCS03c] K. Chaaban, P. Crubillé, and M. Shawky. *SCOOT-R: Middleware communication services for real-time systems*. Principles of Distributed Systems: 7th International Conference, OPODIS 2003, La Martinique, French West Indies, December 10-13, 2003.

[CCS03b] K. Chaaban, P. Crubillé, and M. Shawky. *SCOOT-R: A framework for distributed real time applications*. In proceedings of the WIP of the 24h IEEE Real-Time Systems Symposium, Cancun, Mexico, November 2003.

[CCS03a] K. Chaaban, P. Crubillé, and M. Shawky. *Real-time embedded architecture for intelligent vehicles*. In proceedings of the Fifth Real-Time Linux Workshop, Valencia, Spain, November 2003.

# *Contents*

# *Liste of tables*

# *Liste of figures*

# *Introduction*

In last years, the real-time systems community has been making significant effort in using commercial off-the-shelf (COTS) components to develop performance-assured systems. Several research initiatives have been undertaken to enable writing cost-effective software that provides Quality-of-Service (QoS) guarantees on multiple platforms whose resource capacity, speed and load profile are unknown at application design time.

Today, the notion of distributed real-time architecture is commonly accepted in the industrial sector and the real-time systems are used in many fields such as equipment control in the automotive sector (engine, brake, etc.), aerospace (satellite follow-up, automatic piloting, etc.), railway transport, telecommunication, medical (assistance and monitoring of patients), military (follow-up of missiles trajectories), multimedia (video conference, etc.).

The research works depicted in this thesis have been conducted at *Heudiasyc* laboratory, among the *advanced vehicle* team, with the support of the European project *RoadSense*[2]. The main objective of the RoadSense project was to provide a framework for establishing robust Human Vehicle Interface (HVI) requirements that will drive technology change. RoadSense provides the car manufacturers by analysis tools in order to enable them to define the guidelines for the evaluation of the relevance of the assistance systems of future generations. A certain number of criteria (metrics) was defined describing the reaction of the driver vis-a-vis the driving assistance system, the performances of the driver are thus evaluated, with and without the driving assistance system, in order to evaluate its true benefit to the improvement of safety.

Part of our contribution was the development of a design methodology, realization and deployment of a real-time system having a sufficient robustness required to evaluate the Advanced Driving Assistance System (ADAS). The distributed system is called D-BITE for Driver Behavior Interface Test Equipment. The system is conceived to be open and maintainable. This methodology is based on an (OO) Object Oriented middleware dynamically reconfigurable (SCOOT-R), and reflects the study case chosen by the car manufacturers. The middleware allows the real-time communication between the perception sensors, fusion elements, decision modules, and the Man-Vehicle Interface (Figure 2).

In order to evaluate ADAS functions, D-BITE elaborates driver behavior indicators (or metrics) for safety, comfort and support assessment.

Initially, these computations are made *off-line* (for the RoadSense project), i.e., during an experimental phase, all sensors data and video flows are logged on embedded computers aboard the vehicle, and then the driver indicators are computed off-line in order to apprehend the driver vehicle interaction.

Our proposition later is the *on-line* computation of these metrics. SCOOT-R ensures a reliable development framework to design and compute these metrics using sensors data. The on-line computation of metrics allows the integration of these indicators to the ADAS functions in order to take into account the driving situation and the external vehicle environment. On the other hand, this on-line processing

[2]ROADSENSE: ROad Awareness for Driving via a Strategy that Evaluates Numerous Systems, http://www.eu-projects.com/roadsense/

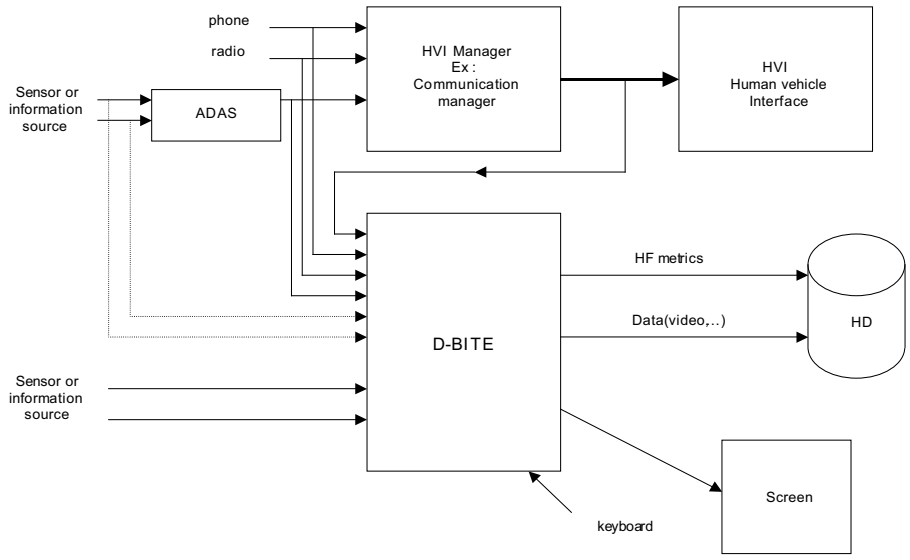



Figure 1: D-BITE architecture

has imposed several requirements concerning for example the real-time and feedback scheduling to adapt the resources to the current driving situation. Computing resources used inside an advanced vehicle are limited, and may fail. Thus, we must adapt the resources according to the criticalness of services and to select them dynamically. For this purpose, we have developed distributed scheduling strategies to deal with asynchronous systems and to facilitate the deployment of such distributed applications.

The thesis effort is therefore divided into three complementary areas: (i) low-level real-time communication support and additional middleware services for synchronisation and dynamic reconfiguration, (ii) distributed real-time scheduling strategies to ensure end-to-end real-time QoS guarantees to applications, and (iii) feedback scheduling techniques for the automotive environment.

## Real-time communication sub-system and middleware services

The main goal of this part was the contribution to the design and implementation of an object oriented middleware called SCOOT-R formerly developed at our laboratory.

The large amount of data to be processed and the distribution of calculation on several computers have lead us to the development of SCOOT-R. It enables programmers to develop distributed real-time applications. It is layered between the application and the OS kernel and handles, on behalf of the application, the *temporal correctness*, *synchronisation*, and *resource allocation* issues in a performance-assured system.

Given the distributed nature of our architecture, the need of a synchronisation mechanism is obvious. SCOOT-R ensures a low-level middleware service for synchronisation between computers. It provides a global time base with high resolution for on-line data time stamping. This reference time is then used to provide a post-processing synchronisation in a distributed architecture.

SCOOT-R is based on client/server and emitter/receiver communication models. It also includes run-time monitoring of real-time constraints and ensures a *dynamic reconfiguration* by replicating software components.

The communication services use the IEEE-1394 bus. The advantage of using this bus is its dual transfer modes, its determinism and high bandwidth capabilities. Moreover, the IEEE-1394 norm provides low level dynamic configuration and synchronisation services that are useful for the design and implementation of reconfigurable distributed real-time systems.

To facilitate the development of communication-oriented services, our communication subsystem is implemented using the RTAI Linux kernel originally developed at the University of Milano. We have chosen Linux RTAI system primarily for its performance and its complete support and availability.

This middleware solution is very relevant for us; instead of losing time and effort to code a real-time kernel, we concentrated on the true added value of our contribution, such as, the simple data objects sharing, synchronisation with high temporal resolution, and distributed scheduling.

## Distributed real-time scheduling

Middleware in a distributed real-time system can provide the framework and mechanisms required to perform the necessary scheduling strategies. Thus, it provides the required end-to-end support for various real-time quality of service (QoS) aspects, such as bandwidth, latency, and jitter.

During the initial development of the SCOOT-R middleware, the management of exchanged messages was relatively rudimentary by using the FIFO scheme for messages and fixed priority for tasks. The initial scheme did not consider the priority of the task sending or receiving the message.

In this phase, the work was devoted to the development of distributed scheduling strategies in order to achieve end-to-end delay predictability. Two scheduling strategies are developed:

- Client Priority Propagation (CPP)
- Client Deadline Propagation (CDP)

In addition and to deal with overloaded situations, we have developed a *hybrid* static/dynamic distributed scheduling strategy called the Best Benefit (CDP-BBA) strategy. It explicitly takes into account both the deadlines and criticalness of tasks and messages while making scheduling decisions.

The main objective of these proposed scheduling strategies is to provide SCOOT-R middleware by a global real-time scheduling service to support the end-to-end timing constraints for the distributed client/server interactions.

The distributed system considered here is composed of nodes interconnected by a deterministic network bus. The target applications are qualified by their timing constraints and they may include functions with different levels of criticalness. Moreover, the proposed scheduling strategies are well adapted to the transactional models (e.g., client/server) and are implemented using our SCOOT-R middleware.

## Dynamic feedback scheduling in automotive environments

Future real-time systems for advanced vehicles control will need to have intelligent and adaptive behavior in order to operate in dynamic and non-deterministic environment characterized by the unpredictable nature of the vehicle, road condition and driving situation. Hence, the criticalness of applications evolve dynamically with the driver environment.

To realize such complex system, two objectives must be met: first, safety and critical services must be guaranteed to provide their results with acceptable quality with respect to their deadlines; second, the utility of the system as determined by timeliness, precision and confidence level must be maximized.

Moreover, the static scheduling for future ADAS functions in automotive applications leads to over dimensioning the distributed computing resources. In such applications, critical and less critical components coexist. The criticalness of certain component may depend on the driving situation, partial failure and redundant component activation

In the case study depicted in chapter 5, we consider a set of driving situation defined at pre-runtime. To each driving situation corresponds a set of metrics. Each metric has a *relative* importance level represented by a utility value or *criticalness*. This criticalness depends on the current driving situation and changes with it. We have proposed to use *feedback scheduling* technique to consider the criticalness of components based on the driving situation. These techniques were first used in the field of systems control. We believe applying them to the embedded automotive field to be a novel contribution.

Thus, the adaptation in our approach is carried out on the *driving situation*, which will further lead to the change of the associated metrics components *criticalness*. Thus the schedule will be adjusted so that it satisfies the desired real-time requirements.

# *Introduction générale*

Ces dernières années, la communauté des systèmes temps-réel avait fait un effort significatif en utilisant des composants sur étagère (COTS) pour développer des systèmes informatiques performants. Un certain nombre d'initiatives de recherches ont été introduises afin d'augmenter la rentabilité du développement des logiciels fournissant des garanties de Qualité de Service (QoS) et s'exécutant sur des plateformes hétérogènes dont les profils de capacité, de vitesse et de charge de ressource sont inconnu au moment de la conception d'application.

Aujourd'hui, la notion de l'architecture temps-réel distribuée est généralement acceptée dans le secteur industriel et les systèmes temps-réel sont utilisés dans beaucoup de domaines tels que le contrôle/commande dans le secteur automobile (moteur, frein, etc.), l'espace (suivi satellite, pilotage automatique, etc.), transport ferroviaire, télécommunication, médicale (aide et surveillance des patients), militaire (suivi de trajectoire de missiles), multimédia (vidéoconférence, etc.).

Les travaux de recherche présentés dans cette thèse ont été réalisés au laboratoire Heudiasyc, au sein de l'équipe véhicule avancé et dans le cadre du projet européen RoadSense.

L'objectif principal du projet RoadSense était le développement d'une méthodologie d'évaluation de l'efficacité de nouveaux systèmes d'aide à la conduite. RoadSense fournit les constructeurs automobiles par des outils d'analyse afin de leur permettre de définir les directives pour l'évaluation de la pertinence des systèmes d'aide de futures générations. Un certain nombre de critères (indicateur) ont été définis décrivant la réaction du conducteur vis-à-vis du système d'aide à la conduite, les comportements du conducteur sont ainsi évaluées, avec et sans le système d'aide à la conduite, afin d'évaluer sa vraie participation à l'amélioration de la sûreté.

Une partie de notre contribution dans le projet était le développement d'une méthodologie de conception, réalisation et déploiement de système temps-réel ayant un fonctionnement suffisamment robuste requis pour l'évaluation des systèmes d'aide à la conduite (ADAS). Ce système distribué s'appelle D-BITE (Driver Behavior Interface Test Equipment) et il est conçu pour être ouvert et maintenable.

Cette méthodologie s'appuiera sur un intergiciel orienté objet dynamiquement reconfigurable, SCOOT-R. Cet intergiciel permettra la communication temps-réel entre les organes de perception, de fusion, de décision et de l'Interface Homme-Véhicule (IHV) (Figure 2).

Afin d'évaluer ces systèmes d'aide à la conduite, le système D-BITE élabore des indicateurs permettant l'évaluation du comportement de conducteur, sa sûreté, son confort et son support.

En premier temps, ces calculs sont faits hors ligne (pour le projet RoadSense), c.-à-d., pendant une phase expérimentale, toutes les données venant des capteurs et des caméras sont enregistrées sur des ordinateurs embarqués à bord du véhicule pour êtres traitées ultérieurement en post traitement.

Notre proposition plus tard est le calcul en ligne de ces indicateurs. SCOOT-R assure un cadre de développement fiable et flexible pour la conception et le calcul de ces indicateurs en utilisant des données capteurs. Le calcul en ligne de ces indicateurs permet leur intégration dans la boucle fermée

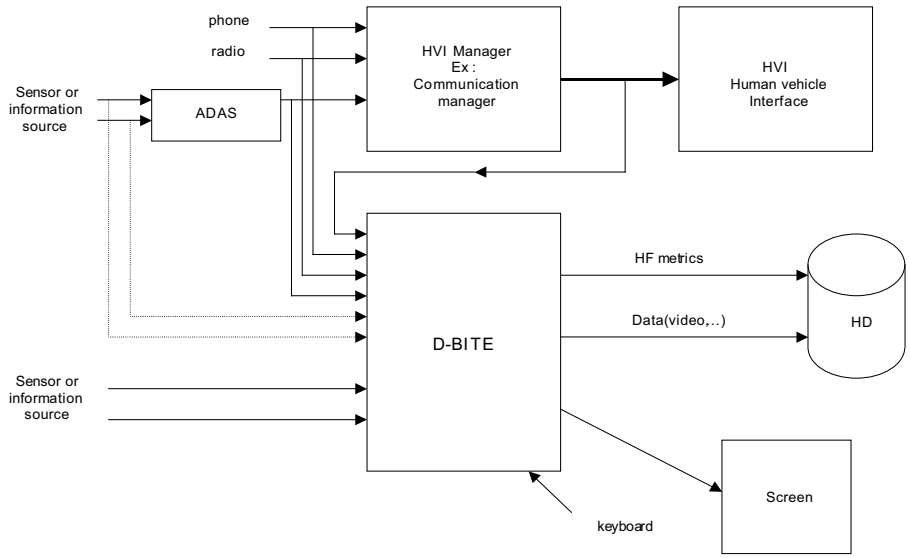



Figure 2: Architecture du système D-BITE

d'évaluation des systèmes d'aide à la conduite ADAS afin de tenir compte de la situation de conduite courante et du comportement de conducteur.

D'ailleurs, ces calculs en ligne et en boucle fermée ont imposés un certain nombre de contraintes sur le système informatique sous-jacent, notamment l'ordonnancement temps-réel et contextuel pour adapter les ressources à la situation de conduite courante.

Etant donné que les ressources informatiques utilisées dans un véhicule avancé sont limitées, et peuvent échouer. Ainsi, nous devons adapter les ressources selon la criticité des services et de les choisir dynamiquement.

Ce document est donc divisé en trois parties complémentaires : (i) le middleware reconfigurable et la communication, (ii) l'ordonnancement temps-réel des opérations SCOOT-R avec un objectif de qualité de service de bout en bout, et (iii) les techniques d'ordonnancement régulé adaptées aux situations de conduite.

## Sous système de communication temps-réel

L'objectif principal de cette partie était la contribution à la conception et à l'implémentation d'un intergiciel orienté objet, SCOOT-R.

La masse importante de données à acquérir et à traiter en temps-réel ainsi que la distribution du calcul sur plusieurs machines nous a motivé pour le développement de SCOOT-R.

SCOOT-R fournit un ensemble de services, construit comme une couche middleware au dessus d'un noyau temps réel. Il permet l'échange d'objets entre des processus s'exécutant sur des machines distribués selon un modèle client/serveur et émetteur/récepteur et en respectant des échéances temporelles strictes.

Parmi les services principaux fournis par SCOOT-R on peut citer ceux de la gestion de répliques pour la reconfiguration dynamique du système et la fourniture d'un temps global pour la synchronisation entre les applications.

Les services de communication utilisent le bus IEEE-1394. Parmi les avantages de ce bus sont

ses double modes de transfert, son déterminisme ainsi que sa bande passante très élevée. D'ailleurs, la norme IEEE-1394 fournit des services de reconfiguration dynamique et de synchronisation bas niveau qui sont utiles pour la conception et l'implémentation des systèmes distribués temps-réel dynamiquement reconfigurable.

## Ordonnancement distribué temps-réel

L'intergiciel dans un système distribué temps-réel peut fournir les outils et les mécanismes nécessaires pour la conception et l'implémentation des stratégies d'ordonnancement temps-réel. Ainsi, il fournit le support d'exécution de bout en bout requis par les différentes approches de la Qualité de Service QoS temps-réel, tels que la bande passante, la latence, et la gigue.

La première version de SCOOT-R ne fournit pas de mécanismes aux clients afin d'indiquer les priorités relatives à leurs requêtes. Néanmoins, cette caractéristique est intéressante, pour réduire au minimum le phénomène d'inversion de priorités, et pour minimiser le temps de latence et le jitter pour les opérations prioritaires. Lors du développement initial de SCOOT-R, la gestion des messages était relativement rudimentaire en utilisant le schéma du premier arrivé premier servi qui ne tenait pas compte de la priorité de la tâche envoyant ou recevant le message. Les tâches sont ordonnancés en se basant sur des profiles de priorités fixes.

Dans cette phase, le travail a été consacré au développement des stratégies d'ordonnancement distribuées afin de minimiser les temps de latence de bout-en-bout. Deux stratégies d'ordonnancement sont développées:

- Client Priority Propagation (CPP)
- Client Deadline Propagation (CDP)

Afin de traiter les situations de surcharge du système, nous avons développé une stratégie mixte d'ordonnancement appelée CDP-BBA (Best Benefit Algorithm). Elle tient compte explicitement des échéances absolues et de la criticité des tâches/messages.

L'objectif principal de ces stratégies d'ordonnancement proposées est de fournir SCOOT-R par un service d'ordonnancement global pour garantir les contraintes temporelles de bout en bout des opérations prioritaires.

Le système distribué considéré ici est composé d'un ensemble de machines reliées par un bus réseau. Les applications cibles sont qualifiées par leurs contraintes temporelles et elles peuvent inclure des fonctions avec différents niveaux de criticité. D'ailleurs, les stratégies d'ordonnancement proposées sont bien adaptées aux modèles de communication transactionnels (par exemple, client/serveur) et sont implémentées mises en oeuvre en utilisant notre intergiciel SCOOT-R.

## Ordonnancement dynamique à boucle fermée en environnements automobiles

Les futurs systèmes temps-réel utilisés pour le contrôle des véhicules avancés devront avoir le comportement intelligent et adaptatif afin de fonctionner dans un environnement dynamique et non déterministe caractérisé par la nature imprévisible du véhicule, état de route et la situation de conduite. Par conséquent, la criticité des applications évolue dynamiquement avec l'environnement de conducteur

Afin de réaliser de tels systèmes complexes, deux objectifs doivent être abordés: d'abord, les services critiques du système doivent être garantis pour fournir leurs résultats avec une qualité acceptable et une fiabilité suffisante; en second lieu, l'utilité du système défini par l'opportunité, la précision et le niveau de confiance doivent être maximisées.

D'ailleurs, l'ordonnancement statique des futurs systèmes d'aides à la conduite (ADAS) mène typiquement au surdimensionnement des ressources de calcul distribuées dans le véhicule. Dans de tels systèmes, des composants critiques et moins critiques peuvent coexister. La criticité de certains composants peut dépendre de la situation de conduite courante, de la défaillance partielle du système et de l'activation des composants redondants.

Dans l'étude de cas représentée en chapitre 5, nous considérons un ensemble de situations de conduite défini à priori. À chaque situation de conduite correspond un ensemble des indicateurs.

Dès que la situation de conduite change, la criticité des indicateurs associés change également. Nous avons proposé d'utiliser la technique d'ordonnancement dynamique en boucle fermée (feedback) pour adapter l'allocation de ressources à la criticité des composants selon la situation de conduite.

# *An overview on real-time systems*


## Abstract

The purpose of this introductory chapter is to describe the environment of real-time computer systems. We start with the definition of a real-time system and a discussion of its functional and temporal requirements. A particular emphasis is made on the temporal requirements. These definitions provide a common base of real-time vocabulary, updated technologically and opened to the embedded applications, in particular the automotive field.


## Contents

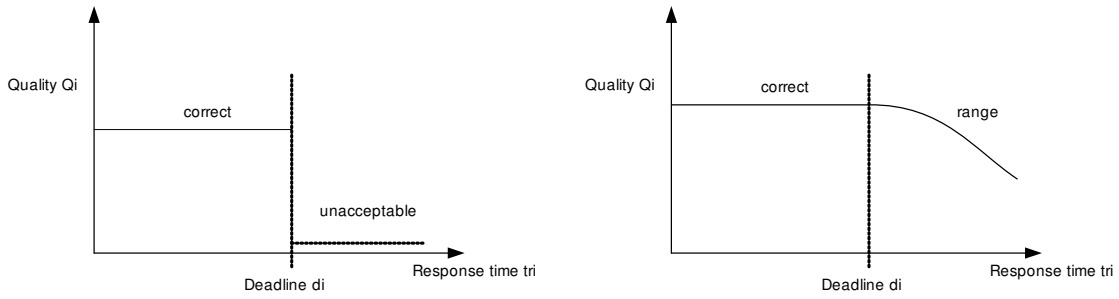
## 1.1 The concept of real-time system

A real-time computing element is always part of a larger system, the real-time system. If the real-time computer system is distributed, it consists of a set of nodes (computers, microcontrollers) that cooperate to realize a set of functions prone to real-time constraints and interconnected by a real-time communication network.

The complexity of processes that must be controlled or supervised, the high amount of data and events to be treated, the geographic distribution of processes at one hand, and the appearance from many years of industrial local buses on the other hand have conducted to reconsider the centralized real-time applications and encouraged the distributed solutions.

Today, the notion of distributed real-time architecture is commonly accepted in the industrial field. Real-time systems are used now in many fields such as equipment control in the automobile field (motor, brake, etc.) [Ack97][AFH+03] [MGFSK04][AFH+03], aerospace field (satellite follow-up, automatic piloting, etc.) [ABA+97][SGHP97], railway train [ZH03], telecommunication [AAS97], medical field (assistance and control of patients) [GEP+00], military field (follow-up of missiles trajectories) [ABA+97], multimedia (video conference, etc.) [PJ90][Jef92].

The design of a hard real-time system, which must produce the results at the correct instant, is fundamentally different from the design of a soft real-time one.




(a) Hard real-time system (b) Soft real-time system

Figure 1.1: Hard vs. soft real-time systems

**Definition 1. Hard real-time systems:** *A* hard *real-time system comprises functions with strict constraints. The principal objective in the realization of such system is to strictly satisfy all the hard time constraints of hard real-time functions. With the situation depicted in Figure 1.1(a), the damage that arises if the deadline is missed is catastrophic and the result is unacceptable. In a hard real-time system, it is necessary to have a static (*a priori*) validation or analysis in order to guarantee the temporal behavior of its functions.*

**Definition 2. Soft real-time systems:** *A* soft *real-time systems is defined as a system when the occasional miss of deadlines does not put the system in difficulty. i.e., the non-respect of deadlines leads to a* correct *result, but with degraded quality. In Figure 1.1(b) the failure becomes more severe as time passes beyond the deadline. A soft real-time system comprises only functions having soft real-time constraints. It can be validated while computing statistically the average response time (multimedia, etc.). Components used to implement a soft real-time system (operating systems, communication networks, etc.) do not have necessarily deterministic response times.*

**Definition 3. Firm real-time systems:** *A firm real-time system [BBL01] can tolerate some degree of missed deadlines provided that this rate is bounded and guaranteed off-line. Thus, deadlines can*

*be missed occasionally but producing a late result is worthless. Nevertheless, the term occasional is so ambiguous that it has no practical meaning for a specification. The extent to which a system may tolerate missed deadlines has to be stated precisely. Assuming that some deadlines can be missed, the way that these missed and met deadlines are distributed is important. An example of a firm approach is the (m,k) firm when the system accepts to miss (k - m) deadlines each k consecutive deadlines.*

In most large real-time systems, not all computational events will be hard or firm. Some will have no deadlines attached and others will merely have soft deadlines. The distinction between hard and soft application may, however, produce hybrid behaviors.

**Scheduler, scheduling, and schedulability**: One of the basic parts of a real-time system is the *scheduler*. It allows the establishment of an order to execute system tasks and to switch from task or process to another by applying a scheduling algorithm. A scheduling algorithm rests on a *tasks model* that defines a set of restrictions and constraints that must conform the tasks, and on a *schedulability test* that permits to verify if the temporal requirements of tasks will be met or not. In a critical real-time system, the *schedulability test* must be verified before execution in order to off-line guarantee the temporal behavior of its tasks.

**Periodic, aperiodic and sporadic tasks**: In the real world, events occur simultaneously and a system that interacts with this real world has to mimic this parallel behavior. This is usually done by building the system as a set of cooperating sequential tasks that interact with each other and with the environment, and that execute continuously. Each of these tasks models a small part of the system and these tasks can be either *periodic* or *aperiodic*. A *periodic* task is executed repeatedly, in a regular and cyclic pattern. *Aperiodic* tasks are usually executed in response to asynchronous events, where a measure of the arrival rate is usually provided. There are also *sporadic* tasks that are invoked at arbitrary times but with a specified minimum time interval between invocations.

**Static and dynamic scheduling**: In real-time systems, it is required to guarantee that the temporal constraints will be met during execution. With design guarantees, also called *static* guarantees, a warrant on task temporal constraints has to be given *a priori*, these *static* guarantees correspond usually to static scheduling policies. In run-time guarantees, also called *dynamic* guarantees, no *a priori* guarantees on the tasks completion times can be given, but at run-time, the system determines whether a given task invocation will satisfy its temporal constraints [Nic98]. So, *dynamic* scheduling policies have been introduced to guarantee temporal constraints at run-time [MMM00][SJK88].

For temporal constraints that are guaranteed at run-time, there are two situations. Whenever a task is accepted for execution, it has to be guaranteed that it will finish by its deadline, or it may be accepted for execution even though it may not finish by the deadline. This leads to two types of schedulers: guaranteed and best-effort. Guaranteed schedulers apply an acceptance test and the tasks are either accepted or rejected. Best-effort scheduling accepts the task for execution but they do not provide a guarantee that the task will meet its temporal constraint.

If temporal constraints that have been guaranteed miss at run-time, it is said that at run-time, a specification violation occurs. This can be due to the software (operating system, applications, etc.) or the hardware (computer, communication network, etc.) do not work as it was assumed. It is therefore required to monitor the behavior of the system during execution and either to stop the system or to introduce some fault-tolerance mechanisms.

**Predictability and safety**: One common misconception of real-time systems is that real-time computing is fast computing. This misconception, among others, has been discussed by Stankovic in [Sta88]. The objective in *safe* real-time systems is to meet the timing requirements, and the property required to do so is *predictability*.

*Predictability* requires some level of determinism [SR90]. The predictability is guaranteed as long

as a set of assumptions of the system hold. The usual assumption framework is that:

- the hardware and operating system have a deterministic behavior;
- the tasks do not execute for more than its pre-computed worst-case execution time;
- sporadic tasks do not re-arrive faster than a minimum interarrival time [Nic98].

Moreover, predictable schedule requires that the information including in the tasks model is defined and sufficient. The minimal information needed is the worst case execution time (WCET) of the tasks. This requires information on how the task is implemented and on the hardware details (for instance, the timing properties of the processor instruction set).

However, *safety* is closely coupled to the notion of risk. In a safety critical system, the situation may indeed be worse with actual damage resulting from an early or missed deadline. A system can be defined to be a safety critical real-time system if the damage has the potential to be catastrophic.

**Real world examples**: As an example of the key mass markets of real-time systems is the field of consumer electronics and automotive electronics. The automotive electronics market is of particular interest, because of its high performance requirements (stringent timing, security, and cost requirements).

After a conservative approach to computer control during the last years, a number of automotive manufacturers now view the proper use of computer technology as a key competitive element in the never-ending quest for increased vehicle performance and reduced manufacturing cost (for example, in a Renault Megane I vehicle, one finds 20 embedded microcontrollers). While some years ago, the computer applications on board focused on non-critical electronics or comfort functions, there is now a substantial growth in the computer control of core vehicle functions (critical), e.g., engine, brake, transmission, suspension control and high-level functions (non-critical), e.g., infotainment, comfort, and driving assistance functions. Obviously, an error in any of the core vehicle functions (critical part) has severe *safety* implications.

Another example of real-time systems markets is the multimedia. The multimedia market is an emerging mass-market for specially designed real-time systems (soft real-time systems). Although the deadlines for many multimedia tasks, such as the synchronisation of audio and video streams, are firm. An occasional failure to meet a deadline results in a limited degradation of the quality of service, but will not cause a global failure.

The emphasis of our research depicted in this document is the *automotive* field. Particularly, we are interested to the design of a distributed real-time system for *high-level* automotive applications (non critical). i.e., the development of a software/hardware architecture allowing the evaluation of driving assistance functions while integrating coherent information computing and distribution.

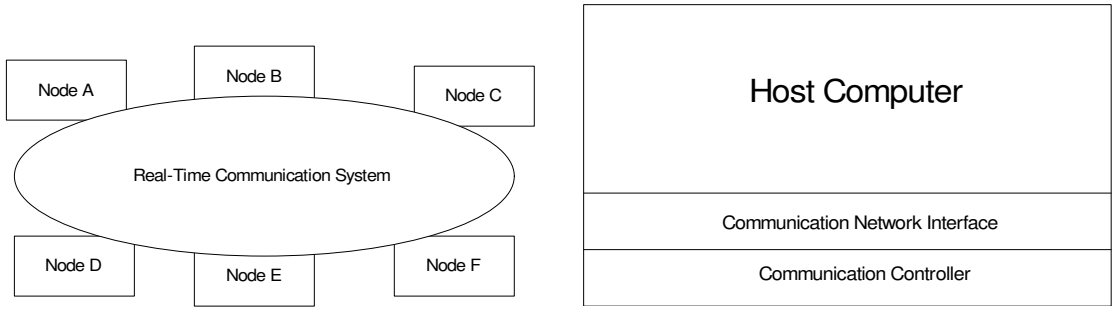
## 1.2 Distributed real-time systems

Distributed real-time systems are now used in many important application areas as mentioned before. Further, distributed real-time technology is becoming increasingly important and pervasive. Strategic directions for research in real-time computing involve addressing new types of distributed real-time systems including globally distributed real-time systems and multimedia systems. Research is required in the areas of system evolution, composability, software engineering, reliability, formal verification, and programming languages. Furthermore, economic and safety considerations, have to be taken into account by these research.

A distributed computer system is a collection of loosely coupled computers that operate autonomously and cooperate on the execution of one (or more) specific problems. Each node is a complete computer system. The nodes have no shared memory and communicate only via messages (Figure 1.2(a)).

A distributed computer system can be viewed as consisting of three major layers: application processes (tasks), a distributed operating system on each node, and a communication subsystem [Kop97b].

In a distributed system, it is feasible to encapsulate a logical function and the associated computer hardware into a single unit node.




(a) Architecture of a distributed system (b) host

Figure 1.2: Architecture of a distributed system

A node can be partitioned into at least two subsystems, the local communication controller, and the host computer (Figure 1.2(b)). The CNI (Communication Network Interface) is located at the transport level of the OSI reference model, and is considered to be an important interface of a distributed real-time architecture.

The application processes are the "user application" in this structure, making use of the services provided by the distributed operating system and the communication subsystem.

The communication subsystem hardware and software facilitate the exchange of messages among the distributed components. Its speed, throughput, reliability, and its behavior regarding eventual failure are determining factors in the performance of the entire distributed system. This is especially true for real-time applications, in which information gathered at one point in the system must be processed and output elsewhere in the system within rigid time constraints. Such systems require dedicated high-performance communications.

The Distributed Object Computing (DOC) is one of the flexible paradigms available today which tackles the development of next-generation distributed and embedded systems [IN02]. The economic constraints with regard to the capacity of computers (processors, microcontrollers, FPGA, etc.) have enhanced the interest in embedded distributed systems and their ability to partition the computing load between their components. Usually, the DOC paradigms are implemented as Object Oriented (OO) *middleware(s)*, i.e., a software layer that extend the functionalities of an OS for the applications.

Middleware hides the architectural complexity of the underlying OS and network and provides the applications by a set of APIs. Thus, the application designers focus on their applications development instead of the adaptation of the low-level services of the underlying OS and network. Furthermore, the middleware extends the basic services of the OS by a rich set of advanced services that are relevant for the development of distributed real-time applications. For example, a middleware provides a synchronisation service in order to guarantee that all the nodes have the same common time base, a

middleware provides also other important services such as a reliable communication service that may tolerate network failures.

Another way that may be used to design a distributed real-time system is the *distributed RTOS*[1]. The distributed RTOS unifies and integrates the control of distributed hardware components (resources), and provides a uniform high-level system interface to the application processes. Thus, it allows the development of an integrated system with a rich set of services and functionalities. Usually, a distributed RTOS is designed to be object oriented, and it has a communication mechanism that allows the transparent use of the resources of a distributed system. Moreover, a distributed RTOS provides certain level of fault tolerance implemented by several policies.

The use of distributed systems is motivated by many reasons, in particular:

- To increase the availability and the run-time maintenance of the computer system by the replacement of a damaged component of the system without significant failure. This aspect is mandatory in the distributed real-time applications for which the failure situations must be rare;
- To process information close to actuators (can operate in safe and degraded mode) and to sensors (decrease of signal to noise ratio). There is a need to adapt the computer system to its environment. It is important that the system takes into account the physical distribution of the real-time applications.
- The demand of computing capacity at lower cost and using COTS-based components;

Distributed systems have some characteristics that distinguish them from centralized systems:

- Communication delays between nodes vary from a message to another according to the network protocol used and the network traffic;
- No common clock; a node has its own time perception and, as the physical clocks derive compared to the time reference, the various times read on the nodes can be different at the same universal time. The computational processes may need references to a common time base, which would lead to unreliable results or errors. Moreover, the scheduling and the computation of real-time constraints of communicating tasks are tributary of these clock drifts and of communication delays;
- Absence of a common memory has as a consequence the impossibility to define a global state using common variables. The impossibility of having an identical global state instantaneously on all the nodes leads to many difficulties to make decisions on the management and the resource sharing.
- Distributed applications are more complex to specify, to design, to implement, to test and to be validated.

In a distributed real-time system, the distributed computer system performs a multitude of different functions concurrently, e.g., the monitoring of real-time entities, the detection of alarm conditions and the time-stamping of data collected on several nodes. These different functions are normally executed at different nodes. In addition, replicated nodes are introduced to provide fault tolerance by redundancy. To guarantee a consistent behavior of the entire distributed system, it must be ensured that all nodes process all events and data in the same consistent order. The events time-stamping provides a global time base to establish such a consistent temporal order.

[1]RTOS: Real Time Operating System

Given the complexity and the distribution nature of our main applications target (assistance driving functions evaluation), we envisaged to use the notion of (OO) object oriented middleware in order to design and implement our distributed real-time system. These applications require certain level of auto-reconfiguration and synchronisation properties. For that, we have developed a middleware support to respond to this requirements by providing a global time-stamping and reconfiguration services of the distributed system.

## 1.3 Real-time operating systems and strategies for distributed systems

The increasing complexity of real-time systems has led to the development of Real-Time Operating Systems (RTOS). RTOSes are now employed widely in many sectors, e.g., military, avionics, must-not-fail telecom, medical, industrial and power plants, railway and automotive equipment.

Unlike a desktop operating system, an RTOS is usually far smaller in size, more modular in structure and focused on the most essential functions. There is no need for an RTOS to include programming interfaces to hundreds of popular software packages that desktop operating systems must provide.

The operating system must be able to take into account this concept of time on all the levels. The following characteristics are thus essential for a RTOS [Gho94][Sta96]:

- A robust programming library (or extension API) that allows a clear specification of real-time applications, guarantees the execution independently of architecture targets and takes into consideration the environment of asynchronous communications;
- A real-time scheduler that applies a scheduling policy (algorithm) to allocate resources while satisfying timing requirements of the underlying tasks model.

In addition, for the distributed real-time operating systems, we may have the additional following characteristics:

- Communication protocols that support the temporal constraints and manage the priority of the real-time messages in order to guarantee the order of execution established between the tasks and their constraints;
- Mechanisms of synchronisation and maintenance of a global real-time clock so that the mechanism of allocation and scheduling uses a single clock in the system. The clocks of the various processors in the system often have a drift, so mechanisms of synchronisation are strongly recommended.

Several operating systems were proposed for real-time and distributed applications. They may be classified by three categories regarding their industrial usage [Tri03]. The first category is the "general-purpose" RTOSes that are usually used in real-time applications having hard and/or soft real-time constraints. The second category is the embedded RTOSes used generally in civil or military applications when a high level of certification is required (e.g., avionic and railway sectors). The third category is the real-time UNIX operating systems, these RTOSes provide applications by POSIX interfaces (Real-time extension of UNIX). These RTOSes provide a rich set of services required by a transactional application such as objects management in a distributed environment.

POSIX is a proposed operating system interface standard based on the popular UNIX operating system. The main objective is to support application portability at the source-code level. POSIX is an

evolving group of standards, each of which covers different aspects of the operating systems. Some of these standards have already been approved, while others are currently being developed.

POSIX.4 is the part of POSIX that defines system interfaces to support applications with real-time requirements. It extends the base POSIX standard by a set of specifications concerning the real-time scheduling, memory management, synchronisation, IPC communication, and timing management functions.

In the first category, we will detail hereafter two examples of distributed real-time operating systems, the MARS and SPRING kernels. Among those distributed RTOSes, only the SPRING OS uses mechanisms of dynamic placement of tasks. These operating systems make of a communication system guaranteeing bounded times on the messages transfer a necessary condition for the correct operation of any distributed and real-time application.

For the automotive applications, and giving the increasing number of microcontrollers in automobiles and other complex systems, the incompatibility of control units made by different manufactures due to different interfaces and protocols, and the need of a support of portability and reusability of the application software, all these requirements have lead to the development of the standard OSEK/VDX that was adopted by the European automotive industry and that will be presented later as a typical example of the second category of RTOSes (Embedded RTOSes).

Moreover, many solutions have been developed in the third category of RTOS and that lead to the introduction of UNIX in the real-time market (e.g., RT-Linux, RTAI, QNX, VxWorks, etc.). We will present hereafter the RTAI kernel as an example of such operating systems.

Let's note that we have chosen the RTAI kernel as an operating system to implement our real-time services encapsulated in data objects and distributed upon a middleware layer.

### 1.3.1 MARS system

The MARS project (MAintainable Real-time System) [KDK+89] began at the university of Berlin, then continued at the university of Vienna. The objective of MARS is to propose an architecture of a distributed real-time system tolerating the faults and intended for critical applications. One of the principal properties of MARS is thus naturally the checking, in an analytical way, the guarantee of the temporal constraints of the tasks set. This objective limits obviously the field of application of MARS, since one can analytically check only tasks whose behavior is known *a priori* (i.e., periodic tasks or who appear at moments known in advance).

An application, according to the MARS approach, can be decomposed into a whole of autonomous subsystems (clusters). A common time base is available on all the subsystems. A subsystem is composed of fault-tolerant units (Figure 1.3). A fault-tolerant unit is an aggregation of three processors functioning in active redundancy to provide a reliable service to the subsystem. These units are inter-connected by an Ethernet bus duplicated and managed by the TDMA (Time Division Multiple Access) protocol. This protocol guarantees messages transmission times known in advance and it is implemented by the MARS kernel atop the Ethernet bus. The messages are the only way to communicate between processing units. The messages are time stamped at the emission and the reception phase, which makes it possible to synchronize the clocks.

The tasks are executed in a cyclic way or at instants fixed and known *a priori*. All the task data must be ready before running the task. A task is running without interaction with the environment and without synchronisation with the other tasks. At the end of its execution, the task distributes its result to the other entities of the subsystem by state messages. This execution model of tasks, which is restrictive, has as advantage the possibility to off-line computing the execution time of each task.

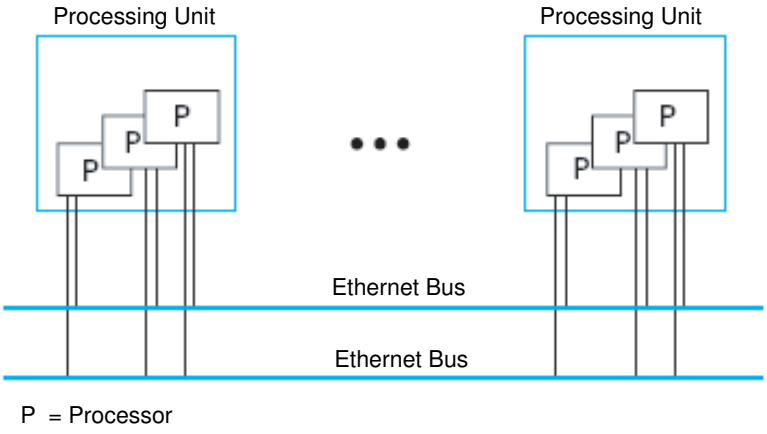



Figure 1.3: Mars architecture

As the tasks periods are known, the communication instants are known *a priori* and the reservations of access time to the TDMA network are carried out *a priori*. Thus, MARS chooses a fixed scheduling of tasks and messages.

In order to take into account certain dynamic aspects (alarms appearance, for example), MARS forces an application design with several functional modes and a fixed schedule must be established for each mode. When a significant event (and envisaged in advance) appears, the system switches the scheduling table.

The MARS kernel is presented here as a prototype kernel using the Time-Triggered (TT) approach for the design of distributed real-time systems. It offers an integrated approach for the system conception, communication, and scheduling.

### 1.3.2 SPRING system

SPRING is a distributed real-time operating system developed at the university of Massachusetts [SR91]. It designed for dynamic, large complex applications. All the types of tasks are considered: periodic or aperiodic tasks, tasks with variable degrees of importance (critical tasks, essential tasks and nonessential tasks), with or without resources and precedence constraints. SPRING also makes it possible to guarantee a specified deadline for a group of tasks (in this case, one is interested in the termination instant of the last task of the group and not only at the termination instants of each task).

Each node of a SPRING system is composed of a multiprocessor machine, which contains one (or several) application processor and one (or several) system processor and subsystem of inputs/outputs (Figure 1.4). The system processors execute the system operations (for example the scheduling algorithm, memory allocation, etc.), in order not to load the operating system by the application tasks. To calculate the execution time of a task, one is interested only in the proper computing of the task and thus all uncertainties concerning the execution times of the system primitives are avoided. The overhead generated by the operating system is often a crucial point in the determination of the execution time to be allocated efficiently to a real-time task. In practice, this time is considered as negligible, or a maximum value is used to avoid all the inconveniences with the applications tasks. SPRING thus brings an effective solution (but expensive) to this problem.

The application tasks are generally executed on the application processors, but can also use

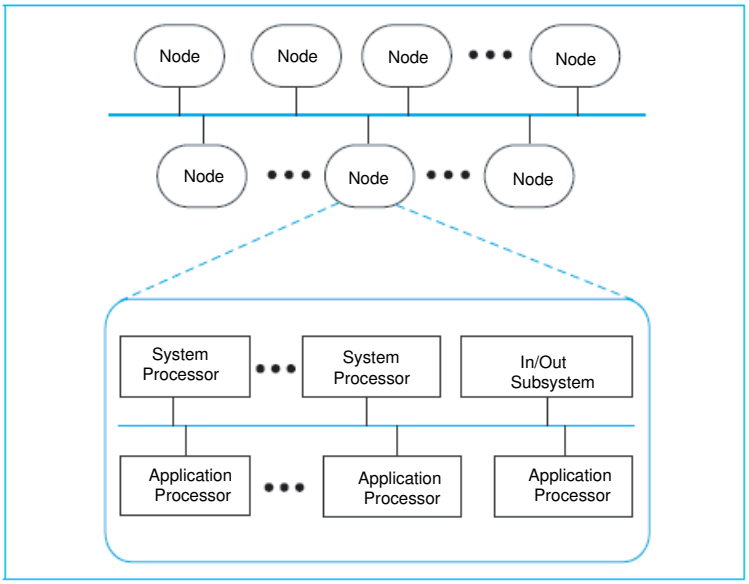



Figure 1.4: Spring Architecture

computing time on system processors. A dynamic algorithm is used to guarantee the constraints of the critical or essential tasks. When the local processors are too loaded to guarantee the constraints of all the local critical or essential tasks, a distributed scheduling algorithm is launched to determine reception nodes for certain tasks. To minimize the transfer times of tasks during the migration operations, SPRING chooses the tasks duplication on the nodes of the network. SPRING uses a dynamic placement with tasks migration. Information on the load of the various nodes is collected and maintained by each node (as a distributed management of the load information).

On the opposite of MARS, SPRING chooses a flexible combination of on-line and off-line techniques: the periodic tasks (which are critical or essential) are treated off line and the sporadic tasks are treated on line. Thus, SPRING has a large variety of techniques to schedule distributed real-time applications but with complex hardware architectures.

### 1.3.3 OSEK/VDX

OSEK/VDX [ZPS+99] is a joint project of the automotive industry that aims to the definition of an industry standard for an open-ended architecture for distributed control units in vehicles. The objective of the standard is to describe an environment which supports efficient utilization of resources for automotive control unit application software. This standard can be viewed as a set of API for real-time operating system integrated on a network management system that together describes the characteristics of a distributed environment that can be used for developing automotive applications.

Originally, OSEK was targeted as a standard open architecture for automotive Electronic Control Units (ECUs) distributed throughout the vehicle. However, the resulting standard is generic and does not limit usage to an automotive environment. Consequently, this standard can be used in many stand-alone and networked devices, such as in a manufacturing environment, household appliances, intelligent transportation system devices, and so forth.

There is now four principal specifications of the standard OSEK. These specifications are the operating system (OSEK-OS), communication (OSEK-COM), network management (OSEK-NM), and implementation language (OSEK-OIL). Recently, the OSEK standard has provided the specification

of OSEKTime OS, a time triggered OS that can be fully integrated in the OSEK/VDX framework;

In the OSEK OS, tasks can be basic or extended and preemptive or non-preemptive. The primary difference between a basic task and an extended task is whether the task can go into a waiting state (in which it is waiting for an event to occur). Only extended tasks can wait for an event. Basic tasks must run to completion unless preempted. Preemptive tasks can be preempted by a higher-priority task becoming ready to run or by an interrupt. Non-preemptive tasks can only be preempted by an interrupt (unless interrupts are disabled).

The tasks are scheduled (nor or full or mixed preemptive scheduling) according to their user assigned priority. The priorities of tasks are statically assigned by the user (the user cannot change tasks priorities at the execution time). OSEK being a specification (with several implementations), requires that any implementation provides at least 8 priority levels.

OSEK-COM specifications comprise an agreement on interfaces and protocols for in-vehicle communication. The in-vehicle communication term means both communication between nodes and internal communication in a node of the whole vehicle. The basic idea is to provide a standardized API for software communication that is independent from the particular communication media used in a way to ease porting of applications between different hardwares.

OSEK-COM provides a rich set of communication facilities but it is likely that many applications will only require a subset of this functionality. For that reason, the standard defines a set of conformance classes to enable the integration of OSEK-COM in systems featuring various levels of capabilities in a scalable way, enabling the car manufacturers to integrate software parts produced by different suppliers.

The communication in OSEK-COM is based on messages. Senders and receivers of messages are either tasks or interrupt service routines (ISRs) in an OSEK OS.

A receiving message object can be defined as either *queued* or *unqueued*. A message received by a message object with the property "queued" (queued message) can only be read once (the read operation removes the oldest message from the queue), like a FIFO (first-in first-out) queue. A message received from a message object with the property "unqueued" (unqueued message) can be read more than once; it returns the last received value each time it is read.

There are three different transmission modes for the messages transmission: Direct, Periodic and Mixed. The message transmission with the *Direct* transmission mode is initiated by the application using the "SendMessage" transfer function (on demand of the application). In *Periodic* transmission mode, the transmission is performed by repeatedly calling the appropriate service "SendMessage" in the underlying layer. Finally, in the *Mixed* transmission mode, the message transmission is performed periodically or occasionally by the detection of relevant variable modification.

The temporal monitoring of communication is performed by watchdogs associated to system alarms. Thus, a deadline expiration, that launches system alarm, may be compensated by a task activation or an event notification to an applicative task.

### 1.3.4 RTAI

RTAI (Real Time Application Interface) is one of the Linux solutions to design and implement real-time applications [Yag01]. It started at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza. RTAI evolved from NMT-RTLinux (New Mexico Institute of Technology's Real-Time Linux), and takes a unique approach of running Linux as lowest priority tasks for the RTAI scheduler. Linux only executes when there are no real-time tasks to run, and the real-time kernel is inactive. The Linux OS can never block interrupts or prevent itself from being preempted.

Basically RTAI consists of an interrupts redistributor; it collects the peripherals interrupts and retransmits them if necessary to Linux. It is not a radical modification of the standard Linux kernel. It uses the concept of HAL (Hardware Abstraction Layer) to isolate some fundamental data and operations (Figure 1.5). Typically, this data information concerns essentially the Interrupts Descriptor Table (IDT), interrupts masking/unmasking functions, and timer management.

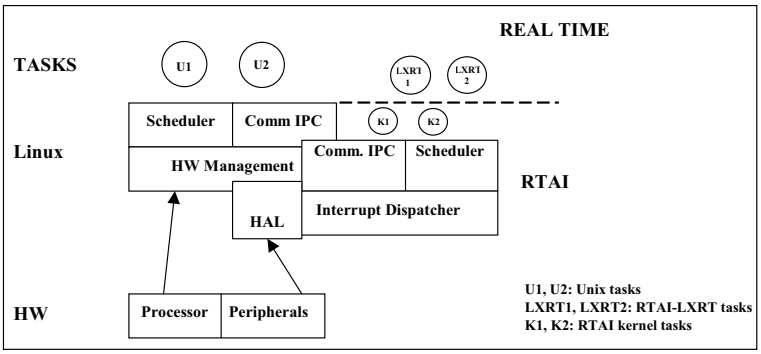



Figure 1.5: RTAI architecture

The RTAI philosophy is to let Linux do all that it can do well. For example the initialization of the system and management of the peripherals (which are not concerned by the real-time application). It is based on the mechanism of "loadable modules" of Linux to install the real-time services of RTAI. Loading a real-time module is not a real-time operation and thus Linux can do it. This makes it possible to use all the services of the Linux kernel during the initialization phase of RTAI module.

RTAI provides special system calls to implement periodic threads and there are two scheduling policies available, (FIFO) RT_SCHED_FIFO and (Round Robin) RT_SCHED_RR. Being an open source kernel, RTAI allows the developers to define and implement their own scheduling policies in the native kernel. There are now several scheduling algorithms that are developed for the RTAI kernel (e.g., DM, EDF, etc.).

RTAI provides also FIFOS and Shared Memory which are used to communicate between real-time tasks in kernel space and Linux applications and also between real-time tasks.

An important extension of RTAI kernel is the LXRT (LinuX Real-Time). It offers soft and hard real-time functionality to Linux user space tasks. The main objective is a fully "symmetric API", i.e., to offer the same real-time API to user space tasks (LXRT1, LXRT2 tasks in Figure 1.5) as what is available to RTAI kernel tasks (K1, K2 tasks in Figure 1.5). A symmetric API, available in user space, reduces the threshold for new users to start using real-time in their applications, but it also allows for easier debugging when writing applications.

## 1.4 Communication networks

Distributed real-time applications impose temporal constraints on communicating tasks execution; these constraints are reflected directly on the messages exchanged between the tasks when those are placed on different nodes. In a distributed real-time application, tasks may have strict or soft temporal constraints, and consequently the exchanged messages may have strict or soft constraints also. For example, a threshold exceedance detection message must be transmitted and received with strict constraints before it generates a failure, whereas a file transfer in general does not require strict temporal constraints.

The communication stack must provide a mean to express the temporal constraints of messages and must implement protocols that guarantee the respect of temporal constraints specified in the services.

Several works have been developed in real-time networks for packet switching networks and for local area networks with multiple access. In the first category of networks, efforts were primarily devoted to ATM (Asynchronous Transfer Mode) to take into account the temporal constraints in the multimedia applications [SK96][C.96][VZF91]. In the second category of networks, the work primarily concerned networks of type CSMA (Carrier Sense Multiple Access) with its various alternatives CSMA/CD (with Collision Detection), CSMA/CA (with Collision Avoidance) (numerous variants of bit-level synchronous bus as CAN, FIP, etc.), CSMA/DCR (with Deterministic Collision Resolution) (e.g., IEEE-1394), Token ring, FDDI (Fiber Distributed Data Interface) and FIP (Flow of Information Process) [ea93][CLW91][MZ95].

Another paradigm of networks was introduced for safety-critical distributed real-time control systems called the time-triggered networks (MARS internal network, TTCAN and Flexray [Kop00]). In a time-triggered architecture all information about the behavior of the system, e.g., which node has to send what type of message at a particular point in time, is known a priori (at design time) to all nodes of the network. Thus, the architecture ensures a high level of determinism for their applications. This paradigm of networks will be presented more in details in the next chapter (section 2.4.2.2).

In a packet switching network, each node connected to the network is regarded as a subscriber and does not know the protocols used inside the switching network. To transmit its data, each subscriber establishes a connection according to a contract guaranteeing a certain quality of service (loss rate, maximum transfer time, etc.). The nodes (or subscribers of the network) can neither enter in competition with others, nor consult each other to know which must transmit data. The temporal constraints are entirely supported by the network, provided that each node negotiates a sufficient quality of service to take into account the characteristics of the messages which it wishes to transmit. Consequently, the mechanisms used are established in the commutation nodes of the network and not in all of them.

In the case of multiple access network, the nodes connected to the network control the medium access via a MAC technique (Medium Access Control) established on each node. Each node reaches the shared medium either by competition, or by consultation (by using a token, for example) according to the type of the MAC technique used by the network. If a node sends a packet on the medium, it will be directly received by its recipient (except, obviously, in the event of collision or of network using equipment of interconnection such as bridges and routers). The nodes must be configured (in particular by determining the priorities of the messages or the nodes, times of use of the tokens, etc.) to be able to guarantee the temporal constraints imposed on the messages. Consequently, the associated mechanisms are implemented on the nodes.

For the automotive field, additional requirements for future in-car control applications include the combination of higher data rates, deterministic behavior and the support of fault tolerance, reliability and availability are beginning to add specifications to the communication technology that are not currently addressed by existing communication protocols (e.g., CAN). FlexRay protocol has been specified as a route to a new network standard that offers high bandwidth, fault-tolerant operation and deterministic behavior as a basis for advanced future automotive applications, such as steer-by-wire and brake-by-wire. The FlexRay data rates of up to 10 Mbps open up new dimensions of automotive communication. Two redundant communication channels support fault-tolerant operation. Unlike event-triggered systems such as CAN, FlexRay is based on a time-triggered architecture where communication is organized in predefined time slots on the FlexRay bus. This ensures deterministic behavior with predefined latencies and avoids bus overloads.

An important subcategory of the "multiple access network" is the infotainment buses. In such buses,

the messages transfer delays are bounded thanks to a fairness MAC protocol. Another important service ensured by such networks is the synchronisation technique and the dual transfer mode (asynchronous and isochronous modes). The network offers to nodes a global time base synchronisation. Such buses have typically large bandwidth and high reconfiguration capabilities. Examples of such network buses are the MOST bus [Gro00] and the IEEE-1394 bus [And98]. This kind of networks will be discussed in details in the next chapter (§2.4.2).

A real-time communication is defined by explicit temporal constraints, i.e., it must begin or finish in a specified time interval. Usually, the temporal constraints associated to a real-time message are of two types: bounded transfer time and bounded jitter. The networks that have the adequate mechanisms to guarantee the respect of these constraints are called real-time networks, also called field buses.

Many works have listed the main properties that real-time networks must have [PD93][Ram87]:

- prediction of messages response times: based on the media access strategy used, a network can be able to guarantee the respect of the temporal constraints of messages;
- known maximum limit for the transfer time and the jitter;
- high degree of schedualibility, i.e., guarantee *a priori* the respect of temporal constraints of a high number of messages with the possibility of periodic or aperiodic messages exchange;
- possibility of messages exchange with strict temporal constraints;
- protocol robustness, i.e., its aptitude to continue to work correctly in the presence of variations of the traffic and overloads;
- good performance and bounded CPU and memory usage;
- transmission error and link breakdown detection and/or fault tolerance;
- high level of validation and testability.

In traditional networks with packet switching, the performances are measured primarily in terms of average response time. In real-time networks, it is necessary to be able to guarantee *a priori* bounded transfer times for individual messages (Figure 1.6). Therefore, a network with high bandwidth is not necessarily a real-time network. It is more particularly by the media access strategy, and not by its flow, that a network can be able to guarantee the respect of the temporal constraints of messages. A network must however have minimal performances required by the target applications.

The communication protocols play a significant role in the distributed real-time applications. Moreover, many current works concern the manner of modifying or extending the OSI (Open Systems Interconnection) model to adapt them to the real-time. The introduction of temporal constraints imposes the integration of new mechanisms, in particular the messages scheduling [CDKM00].

Given the high bandwidth, dynamic reconfiguration, and synchronisation requirements of our high-level in-vehicle applications, we have chosen the IEEE-1394 bus as a communication media of our distributed system. Moreover, its ability to simultaneously stream multiple channel of audio and high-quality video and also the handling of asynchronous transactions make it the technology that can be used for the infotainment and multimedia functions, as for the assistance driving systems in the vehicle. The IEEE-1394 bus will be presented in this document in section 2.4.2.3.

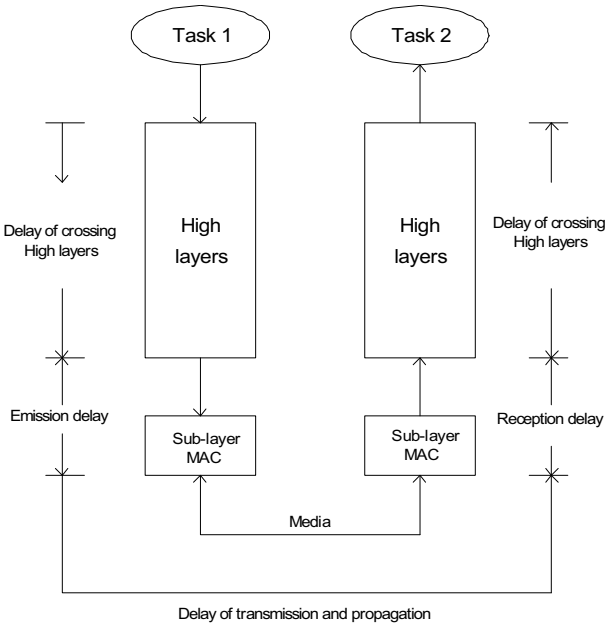



Figure 1.6: Delay intervening in a real-time communication

## 1.5 Conclusion

Real-time systems may be divided into two categories: *hard real-time* and *soft real-time* systems. Hard real-time are those real-time systems in which the time constraints of the processing requests play critical role; that is not meeting the constraints of an accepted processing request is considered a *system failure*. A soft real-time system, on the other hand, may be desirable to meet deadlines, but failure to do so does not cause a system failure.

The next generations systems are expected to be large, complex, and *distributed*. A distributed system will be constituted of a set of nodes connected by a *communication network*. Several works have been developed in real-time communication networks for *packet switching networks* and for local *area networks with multiple access* (e.g., fieldbus). Industrial fieldbus would allow low cost and simple devices to be interconnected to form a network with real-time characteristics.

However, most of the research on real-time systems has been concentrated on *scheduling algorithms* and *operating systems* which contain many of these algorithms.

The next chapter will present the notion of middleware as a solution to facilitate the design and development of distributed real-time systems.

# *From Middleware To Real-Time Automotive Middleware*

## Abstract

In the chapter 1, we have presented an overview on distributed real-time systems by outlining their main functional and temporal features. This chapter presents the middleware technology as a solution to facilitate the development of distributed and real-time systems.

Today, the benefits of middleware are desirable not only for the software development and transactional database but also for the development of distributed, real-time, and embedded systems that impose stringent quality of service (QoS) constraints (e.g., bandwidth, latency, and dependability).

Recently, several initiatives from car industries and third-part suppliers have been taken for the definition of an automotive communication middleware as a software architecture, shared between them and ensuring the portability and interoperability of the in-vehicle applications.

### Contents

## 2.1 Introduction

Middleware is a class of software frameworks designed to help manage the complexity and heterogeneity inherent to distributed systems. It is defined as a software layer above the operating system but below the application program that provides a common programming abstraction across a distributed system (Figure 2.1). Middleware provides higher-level building blocks for programmers than OS Application Programming Interfaces (APIs) [Bak03]. This significantly reduces the burden on application programmers by relieving them of this kind of tedious and error-prone programming. Middleware is sometimes informally called "plumbing" because it connects parts of a distributed application with data pipes for message exchange.

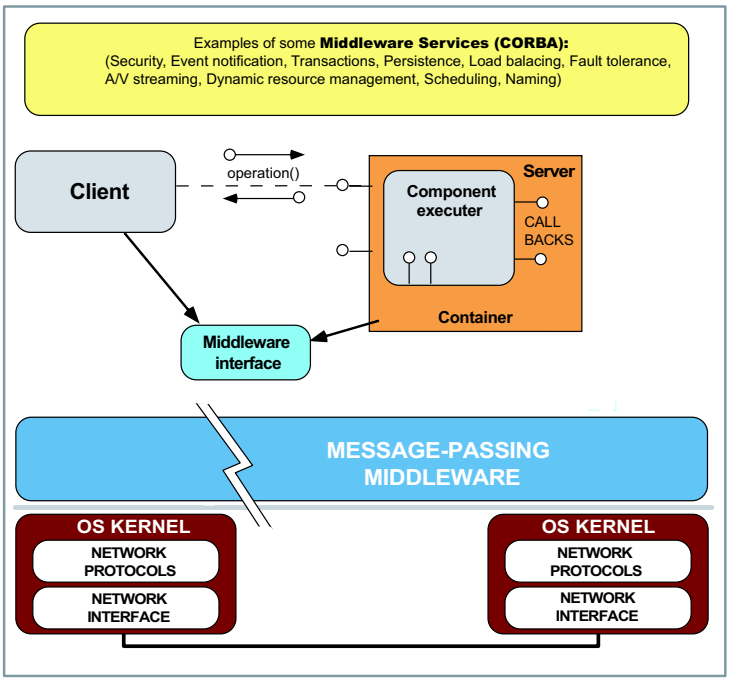



Figure 2.1: Middleware architecture (CORBA middleware example)

As shown in Figure 2.1, a middleware is built above an operating system using its interfaces and its low-level communication stack. The middleware adds also high-level communication interfaces and provides usually a generic object-oriented API for a component-oriented model.

Moreover, it offers a set of common services that are not provided by the underlying operating system. Thus, the time available for providing new functionalities is decreasing. Often this can only be achieved if components are got off-the-shelf and integrated into a system rather than built from scratch. Components to be integrated may have incompatible requirements for their hardware and operating system platforms; they have to be deployed on different hosts, forcing the resulting system to be distributed [Emm00].

Hereafter, we describe the general-purpose middleware(s) (§2.2) by presenting their main categories and features of their software architecture. Then, we introduce and define the notion of real-time middleware (§2.3) by presenting some typical examples. We end this chapter by a presentation of the software methodology and architecture for distributed real-time automotive applications (§2.4).

## 2.2 Middleware for information systems

This section gives an overview of the information systems middleware. We begin by the classification of middleware(s) and then we present some characteristics and features that they must fulfill.

We commonly classify middleware systems along several criteria. The following list is not exhaustive, but still shows that many different types of middleware are possible and necessary to solve the integration problems.

### 2.2.1 RPC vs. asynchronous messaging

At an abstract level, remote procedure calls enable programmers to invoke (distributed) services as if they were intra-application procedure calls [Vin02]. Much like function or procedure calls in traditional programming languages, RPCs block the caller's execution while the invoked service carries out the caller's request. In other words, while the called service is busy handling the caller's request, the calling thread stops executing and waits until the request either returns normally or encounters an error such as a timeout condition.

Messaging systems, on the other hand, are based on a queuing model in which producers post data to queues for consumers to retrieve and act upon. Messaging systems are typically data or document oriented, while RPC systems are procedure or object oriented.

### 2.2.2 Distributed object middleware

Distributed object middleware provides the abstraction of an object that is remote, yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques, encapsulation, inheritance, and polymorphism available to the distributed application developer [Bak03][Vin93].

CORBA represents a standard solution which implements the Distributed Object Computing (DOC). Since its first publication in 1991, CORBA specification has provided abstractions for distributed programming that have served as the basis for a variety of distributed systems. Despite its original flexibility and applicability to various environments, CORBA has had to evolve to remain viable as a standard for distributed object-oriented applications [SK00][SGHP97].

There are other examples of distributed object middleware, such as Microsoft DCOM and Java RMI. DCOM is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) and Component Object Model (COM). DCOM's distributed object abstraction is augmented by other Microsoft technologies, including Microsoft Transaction Server and Active Directory. DCOM provides heterogeneity across languages but not across operating system or tool vendors. COM+ is the next-generation DCOM that greatly simplifies the programming of DCOM.

Moreover, Java has a facility called Remote Method Invocation (RMI) that is similar to the distributed object abstraction of CORBA and DCOM. RMI provides heterogeneity across operating system and Java vendor, but not across language. However, supporting only Java allows closer integration with some of its features, which can ease programming and provide greater functionality [Raj98].

### 2.2.3 Middleware and Quality of Service (QoS)

There is no one common or formal definition of QoS. Initially, QoS was used as a *networking term* to describe the speed and reliability of data transmission (e.g., throughput, transit delay, and error

rate). However, the notion of QoS has been extended from the communication layer up through the intervening architectural layers to the application level. In the simplest sense, Quality of Service (QoS) is a framework that permits to formalize and specify the applications's functional and behavioral requirements [BGM+97][AS99].

QoS for distributed real-time systems provides quality of services assessments to tasks (execution) and messages (transfers). The essential QoS mechanisms when sharing resources concern the tasks scheduling, and the scheduling of messages and the communication protocols for the underlying network. QoS performances like delays, losses, incorrect values, etc. are the metrics to be addressed for integrated QoS in distributed real-time systems.

QoS management at the middleware and application levels aims to control attributes such as response time, availability, data accuracy, consistency, and security level [Gei01]. Middleware is particularly well-suited to express QoS at an application program's level of abstraction. Distributed object middleware is particularly well-suited for this due to its generality in the resources it encapsulates and integrates.

Providing QoS to applications can help them operate acceptably when operational patterns or available resources vary over a wide spectrum and with little predictability. This makes the environment appear more predictable to the distributed application layer, and helps the applications to adapt when this predictability is impossible to achieve.

Moreover, QoS in a middleware may be implemented as a software layer inside the software framework, or it may be implemented as a common service for the high level applications.

### 2.2.4 Programming models

Computer scientists have sought to determine the appropriate programming abstractions, particularly for distributed processing and middleware. In this section, we present two main paradigms of programming approaches commonly used by the middleware(s).

#### 2.2.4.1 Client/server model

The client/server model has been the predominant abstraction for building distributed open systems. The client, which binds to a server, initiates the interaction, sends a request, and awaits the answer. In principle, this is a sequential pull model with a single logical control thread. The server stores long-term state information related to particular client/server associations. The term client/server is generally synonymous with distributed systems. However, a significant part of application scenarios fit poorly with the client/server interaction model.

Thus, using the client/server model is inappropriate for interaction scenarios such as multimedia data flow. These application scenarios require other models and more adequate terminology.

#### 2.2.4.2 Publish/Subscribe model

The publish/subscribe paradigm [Gei01] is a model where publishers produce events and subscribers consume events. A subscriber would initially register an interest in a particular event or pattern/set of events through some subscription mechanism. Any subscriber that has expressed an interest in an event will be notified once a producer has generated an event that matches the subscription required by the consumer. Therefore, the publish/subscribe model relies on an event notification system forming the

basis of the underlying transport system. Hence, an efficient mechanism is required in which consumers can register a subscription for an efficient delivery of events between consumers and producers.

The publish/subscribe model differs from other distributed programming models through the decoupling of the event service. This allows for publishers and subscribers to be classified into three categories of space decoupling, time decoupling and flow decoupling. Space decoupling is when there is no reference in a subscriber to any particular publisher.

There are many variants of Publish/Subscribe systems, such as producer/consumer, emitter/receiver, etc. Each of which have a number of differences and capabilities [EFGK03][Emm00].

### 2.2.5 Software architecture

This section presents briefly some trends of the software architecture features and requirements. These features include the middleware layers, transparency of distribution, the dynamic reconfiguration, and the errors detection and recovery.

#### 2.2.5.1 Middleware and layering

A middleware system may be decomposed in multiple layers of middleware [Bak03]. Usually, a middleware system is implemented at the "Application" layer in the OSI network reference architecture (Layer 7), however, some parts of it may be also at the "Presentation" layer (Layer 6). Hence, the middleware is viewed as "application" to the network protocols in the operating system (Figure 2.2).

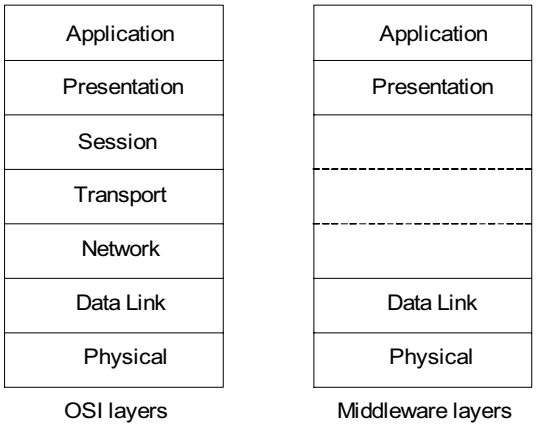



Figure 2.2: OSI and middleware layers

#### 2.2.5.2 Transparent distribution

Achieving transparency to hide the complexity and to isolate applications from the underlying hardware and software details is a cornerstone of all system software, especially for middleware and component-oriented systems [Gei01].

For a distributed component-oriented system, *transparent distribution* means that (1) the functional and temporal behavior of a system is the same no matter where a component is executed and (2) the developer does not have to care about the differences of local versus distributed execution of a component.

The definition and discussion of distribution transparencies played a major role in the International Organization for Standardization's Reference Model for Open Distributed Processing. Distributing transparency is beneficial and necessary for programming distributed applications. However, it cannot be the foremost goal in nomadic computing and context-aware applications. The open research questions involve how to expose network imperfections at the right level of granularity to abstraction and how applications on top of the middleware deal with a selectable degree of transparency.

Increasing awareness of QoS requires making certain effects of distribution explicit. For example, customers who are charged for a certain level of communication service want to know about bandwidth variations or bad transmission quality to optimize the data flow. However, we do not currently have middleware facilities to control the degree of transparency.

#### 2.2.5.3 Dynamic reconfiguration

Dynamic changes in system configuration and operating context at runtime will be inherent characteristics of future computing environments. The purpose of dynamic reconfiguration is to make a system evolve incrementally from its current configuration to another configuration. Dynamic reconfiguration should introduce as little impact as possible on the system execution [AWPvS01]. Moreover, dynamic reconfiguration consists of modifying the configuration of a system during runtime, contributing to the availability of the system.

In this way, systems do not have to be taken off-line, rebooted or restarted to accommodate changes. Changes can be classified with relation to the moment they are envisioned as programmed and evolutionary changes.

New generations of distributed applications often consist of co-operating objects and use object-middleware technology, such as CORBA, Java RMI and DCOM. There are many systems that would benefit from dynamic reconfiguration facilities for object-middleware, such as, critical and/or long-running systems. The development of such systems would be facilitated through the inclusion of (transparent) reconfiguration support in the middleware platform.

#### 2.2.5.4 Detection, diagnosis, and recovery of errors

A component (software or hardware) *failure* is an event that occurs when the obtained service value deviates of the envisaged value by this service. An error is a part of a system's state that may lead to a failure. The cause of an error is called *fault*. A fault may be qualified by its cause and origin such as human, physical, internal, external, conceptual, operational. Another criteria to qualify faults is by their temporal persistence. Faults are generally classified as *transient*, *intermittent*, or *permanent*. *Transient* faults occur once and then disappear. If the operation is repeated, the fault goes away (e.g., *transient* communication failure in a distributed system). An *intermittent* fault may be related to an occasional overload on a node. A *permanent* fault is one that continues to exist until the faulty component is repaired (e.g., shutdown of a node).

If we consider a distributed system built as a set of servers that communicate with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to. Several classification schemes for the *faults models* have been developed. One such scheme is shown in Table 2.1, and is based on the schemes described in [Che99] and [Cri91].

A server's result is considered as *correct* if all its clients receive the same copy and the value of this copy is correct. For an incorrect result, the error is called in the *value domain* if the received value of

at least one client is false. The error is considered as *temporal* error if the value received by at least one client is not in the specified time interval.

The error detection may be implement by *hardware* or *software* solutions according to the errors type. The *temporal* errors may be detected immediately by the client application if the server did not respond in the time interval specified (*Timing failure assumption*). They may be detected also by a *monitor* component that receives regularly specific signalization messages from all the *supervised* components.

Moreover, the value errors may be detected by the comparison bit-to-bit of the results obtained by two identical hardware components. Another part of byzantine errors may detected by the use of one hardware component that performs successively n times the same calculation. The execution is considered correct if the n results are succeeded and identical.

| Type of failure | Description |
|---|---|
| Fail-silent failure | A server stops functioning and produces no ill output |
| Omission failure | A server fails to respond to incoming requests |
| Receive omission | A server fails to receive incoming messages |
| Send omission | A server fails to send messages |
| Timing failure | A server's response lies outside the specified time interval |
| Response failure (or Byzantine failure) | A server's response is incorrect |
| Value failure | The value of the response is wrong |
| Behavior failure | The server deviates from the correct flow of control |

Table 2.1: Failures classification

*Fault tolerance* is the ability of a system or component to continue normal operation despite the presence of hardware or software faults. A fault tolerant technique may be implemented by the *errors* and *faults* processing.

The errors processing is intended to remove errors from the system after or before the error occurs. It includes the *error detection*, *error diagnosis*, and *error recovery*. The *faults processing* aims to prevent the occurrence of old faults that has been occurred in the system. It consists in developing systems in such a way as to avoid that a fault is not activated again.

The *error recovery* may be implemented by several ways [BPB00][DTT99], generally, by *redundancy*. Redundancy is a common practice formasking the failures of individual components of a system. With redundant copies, a replicated object can continue to provide a service in spite of the failure of some of its copies, without affecting its clients. Redundancy may be applied at several levels:

**Information redundancy.** seeks to provide fault tolerance through replicating or coding the data.

**Time redundancy.** achieves fault tolerance by performing an operation several times. Timeouts and retransmissions in reliable point-to-point and group communication are examples of time redundancy.

**Physical redundancy.** deals with devices, not data. We add extra equipment to enable the system to tolerate the loss of some failed components. RAID disks and backup name servers are examples of physical redundancy.

In distributed systems, the best known replication techniques are *active*, *passive*, and *semi-active* replication. Each of these techniques has its own advantages, and they are thus complementary. A brief description of these replication techniques is given below.

**Active replication.** It is a general protocol for replication management that has no centralized control. All copies of the replicated object play the same role: they all receive each request, process it, update their state, and send a response back to the client. Since the invocations are always sent to every replica, the failure of one of them is transparent to the client. Active replication requires the operations on the replicated object to be *deterministic* in order to have a consistent shared state between replicated objects. An interesting property of active replication lies in the fact that a crash does not increase the latency experienced by a client.

**Passive replication.** With passive replication one server is designated as the primary, while all other are backups. The clients send their requests to the primary only. The primary executes the request, atomically updates the other copies, and sends the response to the client. If the primary fails, then one of the backups takes over. Unlike active replication, passive replication does not waste extra resources through redundant processing, and permits nondeterministic operations. Furthermore, passive replication requires additional application support for the primary to update the state of the other copies.

**Semi-active replication.** Semi-active replication style is based on the European Delta-4 (XPA) architecture [BBH+90]. This replication style is designed to have some of the benefits of both the *active* replication and *passive* replication styles, including predictable fail over times and deterministic behavior during program execution. Semi-active replication extends active replication with the notion of leader and followers. In the *leader/follower* model, all copies of an object are active, as they all execute the same function. One copy is designated the leader, however, and is responsible for taking all decisions which affect replicate determinism; such decisions are propagated from leader to followers via synchronization messages.

The Object Management Group (OMG), which standardizes CORBA, has addressed many of these application requirements in the Fault-tolerant CORBA (FT-CORBA) specifications. The Fault-Tolerant CORBA (FT-CORBA) [GNSC04] specification defines a standard set of interfaces, policies, and services that provide robust support for applications requiring high reliability. The fault tolerance mechanism used in FT-CORBA to detect and recover from failures is based on entity redundancy. Since FT-CORBA is a DOC middleware standard, the redundant entities are replicated CORBA objects. Although dealing with the semantic incompatibilities between *real-time* and *fault-tolerant* features for a distributed real-time system seems a promising approach. Thus, the ability to engineer a good fault tolerant solution requires tradeoffs that may compromise a distributed real-time system's ability to meet real-time deadlines, and vice-versa.

Unfortunately, the FT-CORBA model for failure detection and recovery emphasizes a certain type of failure, namely *component failure*, which is also called crash failure. In this type of failure the individual component ceases all interactions with its environment. The policies and detection mechanisms in FT-CORBA, such as the use of heartbeats and timeouts, acknowledge this limited view implicitly.

*Fault tolerance* is not our research topic in this thesis. Our developed SCOOT-R environment provides only errors detection and dynamic reconfiguration means for the failures processing and recovery. Actually, we focus on the redundancy management by the activation of software replicated components in a distributed environment (middleware approach).

### 2.2.6 CORBA architecture

The Common Object Request Broker Architecture (CORBA) [GGM97] is a distributed object framework proposed by the Object Management Group (OMG). CORBA supports distributed object-oriented computing across heterogeneous hardware devices, operating systems, network protocols, and programming languages. Figure 2.3 illustrates the CORBA components described as follows.

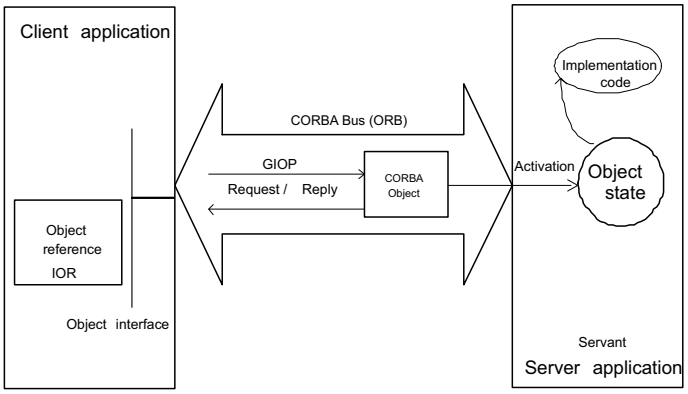



Figure 2.3: Corba Transactions

The Object Request Broker (ORB), the core of CORBA, allows objects to interact transparently with other objects (located locally or remotely). A CORBA object is represented by its interface, is identified by its reference, and is implemented in an object-oriented program as a local object called "servant". The client of a CORBA object acquires the object reference called inter-operable object reference (IOR) and invokes methods on this reference as if the object were located in the client address space.

The dynamic invocation interface (DII)(Figure 2.4) enables clients to directly access the underlying request mechanisms at run time to generate dynamic requests to objects, whose type (interface) were not known at the client compile time. The interface repository provides the type of information that a client needs to dynamically create a request. Similarly, the dynamic skeleton interface (DSI) enables an ORB to deliver requests to a servant that does not have compile-time knowledge of the type of the object it supports. The implementation repository enables late deployment of CORBA objects. It receives the first request targeted to a CORBA object, looks up the object meta information in its database, activates the object, and forwards the request to the target object. Permanent forwarding, in contrast to transient forwarding, also causes automatic forwarding of all future requests from the same client and to the same target object directly from the client ORB. The object adapter activates servants and dispatches requests to them.

The ORB interface provides access to standard ORB services, such as resolving the CORBA initial services (for example the naming service). The general inter-ORB protocol (GIOP) is a standard for inter-ORB communication that enables interoperability among different CORBA-compliant ORBs. The Internet inter-ORB protocol (IIOP) is a particular mapping of the GIOP specification that runs over TCP/IP connections.

CORBA middleware is used today in many areas, such as aerospace, telecommunications, medical systems. However, conventional CORBA suffers from substantial priority inversion and non-determinism, which makes it unsuitable for applications with deterministic real-time requirements. On the other hand, CORBA is hard to be implemented, verified and validated. Moreover, CORBA architecture is very complex for the programmer to validate the code and the specifications of the

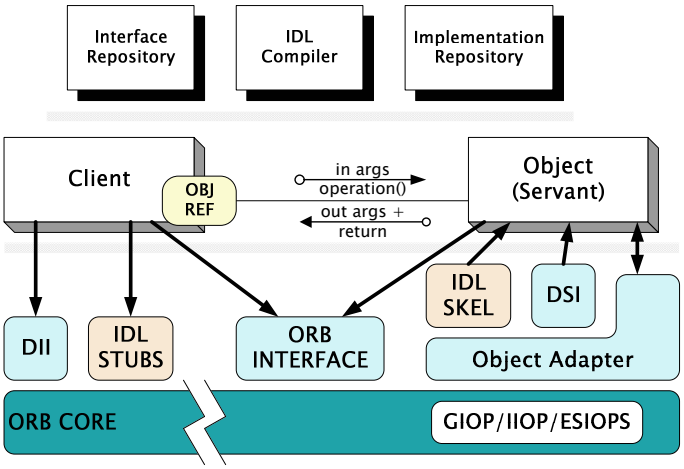



Figure 2.4: Components in the CORBA Reference Model (Client/Server Model)

application. Thus, a real-time adaptation of CORBA for embedded applications was developed and is presented in the next section.

## 2.3 Real-Time middleware

The real-time middleware technology was introduced to consider the end-to-end timing constraints specified by application requirements. The middleware system must allow the user to express the timing constraints, transfer these constraints along the path of execution in the system, and always allocate resources in respect to these constraints.

The field of real-time and dependable middleware is a developing area. It is clear that the development of dependable systems requires standard platforms, just as do other distributed systems. Examples of the real-time middleware include the Real-Time CORBA (RT-CORBA) specification [SK00], and other non CORBA-compliant real-time middleware, such as the ARMADA project [ABA+97], the HADES project [ACCP98], and our own SCOOT-R middleware [CCS04].

For the automotive field, some initiatives from car industries and third-party suppliers for the definition of a communication middleware have been taken. OSEK/VDX [Tri03], the standard of an open-ended architecture for distributed control units in vehicle, has a communication extension (OSEK-COM) whose the main goal is to offer a uniform communication interface through an API to transfer data between tasks independently from their location. Volcano [CRTM99] is a commercial in-vehicle communication middleware used by Volvo. Finally, the remaining initiatives are the EAST-EEA and the Autosar projects that are presented later in this chapter (see section 2.4.1).

In the continuation of this section, we present two examples of real-time middleware. The real-time CORBA (ACE/TAO implementation) and a non-CORBA related middleware (ARMADA middleware).

### 2.3.1 Real-Time CORBA

The Real-time CORBA (RT-CORBA) 1.0 specification defines standard features that support end-to-end predictability for operations in fixed-priority CORBA applications. This specification extends the existing CORBA standard and the recently adopted OMG messaging specification [SK00]. In particular,

RT-CORBA 1.0 leverages features from GIOP/IIOP version 1.1 and the messaging specification's QoS policy framework. All these features and specifications are being integrated into the CORBA 3.0 standard.

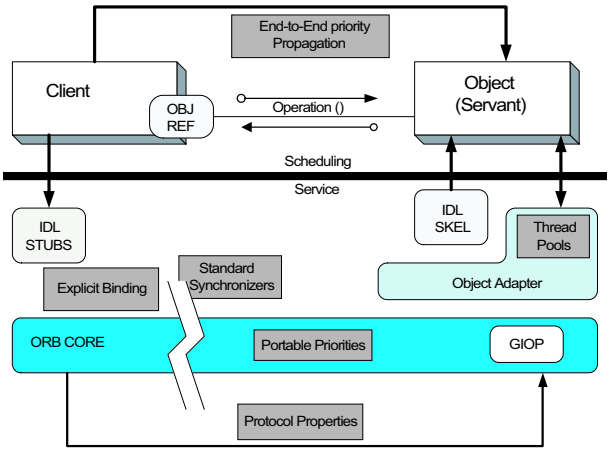



Figure 2.5: Real-Time CORBA

The RT-CORBA specification identifies capabilities that must be vertically (i.e., network interface, application layer) and horizontally (i.e., peer-to-peer) integrated and managed by ORB endsystems to ensure end-to-end predictable behavior for activities that flow between CORBA clients and servers.

These capabilities include standard interfaces and QoS policies that allow applications to configure and control the following resources:

- Processor resources via thread pools, priority mechanisms, intraprocess mutexes, and global scheduling service. This is an encapsulation of the RTOS services to schedule application-level activities end-to-end.

- Communication resources via protocol properties control (e.g., ATM virtual circuits or Internet RSVP) and explicit bindings with non-multiplexed connections. These properties define standard interfaces to allow the control of the underlying communication protocols and endsystem resources. In addition, client applications should *explicitly bind* to server objects.

- Memory resources via buffering requests in queues and bounding the size of thread pools. This model allows server developers to preallocate pools of threads and to set certain thread attributes, such as default priority levels.

Real-time CORBA supports both static [SGHP97] and dynamic [GLS01] real-time scheduling strategies. The *static* real-time scheduling includes essentially the above features to manage CPU by thread pools, end-to-end static priority propagation mechanism, network by protocol properties and explicit bindings, and memory resources by bounding thread pools.

Moreover, in the recently real-time CORBA implementation TAO [GLS01], *dynamic* scheduling strategies define the interfaces for assigning, discovering, and altering the dynamic scheduling parameters. By this way, applications can use the real-time *scheduling service* to specify the processing requirements of their operations in terms of various parameters, such as worst-case execution time, period, end-to-end latency, etc. TAO's run-time scheduler maps these application QoS parameters, to the endsystem OS/network resources.

TAO supports several dynamic scheduling strategies, such as Rate Monotonic (RM), Earliest Deadline First (EDF), Minimum Laxity First (MLF), and Maximum Urgency First (MUF). These scheduling techniques are applied on the CORBA operations (local or remote) that contain QoS parameters related to the application requirements. TAO's scheduler uses the notion of *static* and *dynamic* priorities and subpriorities and according to the scheduling strategy used, TAO'scheduler maps these priorities to the CORBA operations and associated threads and dispatching queues. For example, when an operation request arrives from a client at run-time, TAO's scheduler extracts the QoS parameters of the operation and then apply the local scheduling strategy specified (e.g., RM, EDF, MUF) to the local node resources (CPU and I/O subsystem).

It is important to recall that real-time CORBA does not support the end-to-end EDF. i.e., there is no propagation of the absolute deadlines of operations, there is only the propagation of the relative deadline of operations that will be applied on the remote node (distributed EDF) (see Figure 4.10 for more explication on this point).

As the standard CORBA, the main objective of RT-CORBA is the development of a component-oriented technology and the reuse of application component codes. RT-CORBA is today now in many systems such as multimedia [CW96], telemedicine system [CS01] and embedded system [OSK+00].

### 2.3.2 Armada

ARMADA is a collaborative project between the Real-Time Computing Laboratory (RTCL) at the University of Michigan and the Honeywell Technology Center [ABA+97]. The goal of the project is to develop and demonstrate an integrated set of communication and middleware services and tools necessary to realize embedded fault-tolerant and real-time services on distributed, evolving computing platforms. These techniques offer tools for designing, implementing, modifying, and integrating real-time distributed systems. Key challenges addressed by the ARMADA project include:

- timely delivery of services with end-to-end soft/hard real-time constraints;
- dependability of services in the presence of hardware or software failures, scalability of computation and communication resources;
- exploitation of open systems and emerging standards in operating systems and communication services.

The ARMADA hardware architecture comprises Pentium-based PCs (133 MHz) connected by a Cisco 2900 Ethernet switch (10/100 Mb/s), with each PC connected to the switch via 10 Mb/s Ethernet link.

Ethernet is normally unsuitable for real-time applications due to the MAC protocol (collision detection) and the subsequent retransmissions that make it impossible to impose deterministic bounds on communication delays. However, since ARMADA is conceived to use a private Ethernet, only one machine can send messages at any given time. This prevents collisions and guarantees that the Ethernet driver always succeeds in transmitting each packet on the first attempt, making message communication delays deterministic. In other words, ARMADA uses a high-level Time Triggered protocol whose Ethernet adapter is just a physical layer (With CSMA/CD option inhibited).

The services developed in the context of the ARMADA project are to augment the essential capabilities of a real-time microkernel by introducing a collection of communication, fault-tolerance, and testing tools to provide an integrated framework for developing and executing real-time applications. Most of these tools are implemented as separate multithreaded servers.

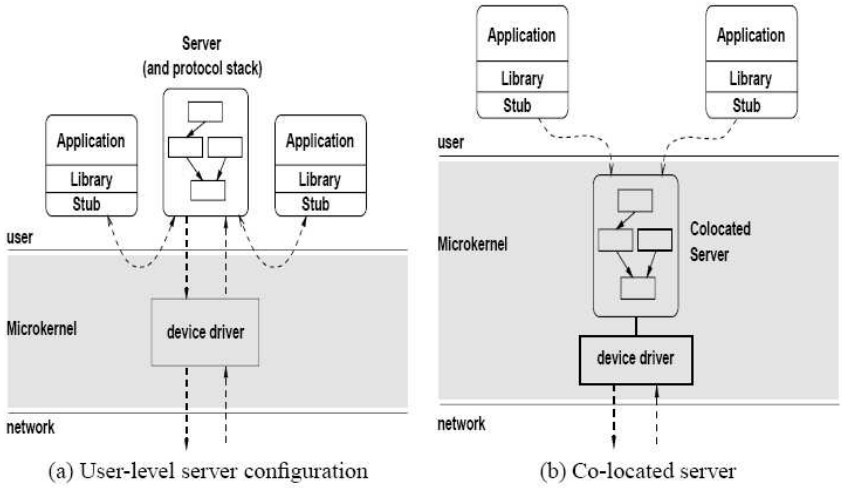



Figure 2.6: Software architecture of armada middleware

ARMADA services are designed as user-level multithreaded servers. Clients of the service are separate processes that communicate with the server via the kernel through a user library (Figure 2.6). A server may be configured to run in user or microkernel space. Whether the server runs in user space or is colocated in the microkernel, processes use the same service API to communicate with it. The library exports the desired middleware API. As operating system, ARMADA uses the MK 7.2 microkernel from the Open Group (OG) Research Institute to provide the essential underlying real-time support for ARMADA services.

The dynamic distributed scheduling and the fault tolerance of ARMADA are the key points. It defines QoS-sensitive scheduling profiles that allow to implement user-level scheduling algorithms.

The tasks are scheduled based on an earliest-deadline-first (EDF) policy layered on top of the underlying operating system scheduler.

Concerning the messages scheduling, ARMADA implements a dynamic priority-based link scheduler at the bottom of the user-level protocol stack. Outgoing real-time packets are scheduled in the order of their deadlines, which is application dependent. The link scheduler implements the EDF scheduling policy using a priority heap for outgoing packets. Best-effort packets are maintained in a separate packet heap within the user-level link scheduler and serviced at a lower priority than those of real-time packets.

After this description of some related and relevant real-time frameworks and realizations. Thereafter, the focus will be made on the in-vehicle real-time systems and the software architecture.

## 2.4 Architecture and methodology for distributed automotive real-time systems

As the emphasis of our research in this thesis concerns the automotive field, we depict in this section software methodology and architecture for distributed real-time automotive applications. We begin by the description of in-vehicle functions going from maintenance and mechanical control functions to high-level functions including infotainment and telematics while passing by diagnosis and *driving assistance functions*. These later functions require a high-level of reconfiguration and monitoring. An

integrated software architecture is needed to design and implement them in an *integrated system*.

Future automotive systems will be required to simultaneously handle multiple safety critical functions and a large number of less critical functions. All of these features are to be provided at a production cost substantially lower than that of current systems, and, at the same time, with a reliability allowing vehicles to be built without mechanical backup systems, even for safety critical subsystems such as braking and steering. In this section, we present some current trends in vehicle networking, together with descriptions of some of the context in which vehicle networks and software architectures are developed today by some car manufacturers.

Functionality in a vehicle is not limited to mechanical control and maintenance functions, but includes also end-user functions. Hereafter, we will outline some important groups of functionality, both supportive and end-user functions that is often addressed in vehicle development.

Feedback control includes functions that control the mechanics of the vehicle, for example engine control and Anti lock-Brake-Systems (ABS). Feedback control systems can be combined to achieve advanced control functions for vehicle dynamics. Examples are electronic stabilizer programs (ESP), and other chassis control systems like antiroll.

The vehicle manufacturers strive to achieve cheaper and more flexible functionality by going towards x-by-wire solutions, such as steer-by-wire, which achieved by replacing mechanical or hydraulic solutions by computer controlled systems.

Discrete control, in this context, includes simple functions to switch on or off devices, e.g., control of lamps or wipers. The challenges for this group of functions often relates to the sheer number of such simple devices and thereby the amount of traffic on the network.

Functions for diagnosis are used in vehicles to support maintenance and vehicle service. Diagnostic functions provide means to investigate physical components, such as sensors, as well as software properties, such as version number and network connectivity. Service functions provide means for updating the electronic system by downloading new software and testing vehicle operation. Because of the large amounts of retrievable information, solutions are needed for automatic diagnostic, or at least tool supported diagnosis.

Infotainment refers to in-vehicle systems related to information and entertainment. Examples are Internet connection and video consoles. This leads to requirements on high bandwidth for vehicle networks. Components like network controllers and software are often purchased off-the-shelf, and must be integrated in a harsh physical environment. Components must also be integrated without impacting safety critical functionality in the vehicle.

Telematics [AFH+03] is a name of the set of functions that uses communication networks outside the vehicle to perform their task. There is a strong trend in the vehicle industry to increase the use of telematics. Examples include fleet management systems, maintenance systems, and anti-theft systems.

### 2.4.1 Embedded automotive architecture: methodology of design

The extra-vehicle functions and modules mentioned above are mostly stand-alone functions although some of the information is already shared. One of the main challenges is therefore the integration of different electronic systems, subsystems, functions and components, delivered by different suppliers into the complete network of a vehicle system. The challenge is to efficiently manage the constantly increasing complexity of electronically controlled functions in today's and tomorrow's vehicles.

Several French and European initiative works have been started during the last years in order to respond to these requirements and research challenges. The first project was the French research action

AEE ("Architecture Electronique Embarquée")[1]. In order to maintain the competitive position of the French car industry, the Ministry for Industry decided to support the project AEE (1998 - 2001). The main goal of the project was to define a unified software framework of the embedded architecture in transport vehicles, to provide this framework by tools of architecture design and analysis, and to facilitate the installation of multi-suppliers processes to develop these architectures. AEE teamed several partners including car manufacturers, equipment suppliers, and academic partners in the real-time field.

The European project EAST-EEA (Embedded Electronic Architecture) [2] has appeared as a continuation of the French project AEE. EAST-EEA began in January 2002 and stopped by the end of 2004. It mainly aims at improving, through a European co-operation, the management of the increasing complexity of the in-vehicle functions controlled electronically in the current and future cars. The major objective of EAST-EEA was to enable a proper electronic integration through the definition of an open architecture aiming to the interoperability of the software and hardware components. For that, a software architecture in layers is aimed using the concept of middleware which offers interfaces and services to support the portability of the embedded software modules with a high level of quality.

In order to continue the European research activities in this direction, the European project AUTOSAR was initiated to pursue AEE and EAST-EEA projects. The objective of the AUTOSAR partnership is the establishment of an open standard for automotive E/E architecture. It will serve as a basic infrastructure for the management of functions within both future applications and standard software modules.

All the above mentioned projects and actions have focused on the effort of standardization and specification of the in-vehicle embedded architecture and associated services. Thereafter, we present some realizations for the in-vehicle software architecture development.

#### 2.4.1.1 BASEMENT

BASEMENT is a real-time architecture for in-vehicle applications developed within the Swedish Road Transport Informatics Programme project Vehicle Internal Architecture (VIA). The objective has been to design a platform that meets the stringent demands of the automotive industry [HLB+97].

BASEMENT covers application development, as well as the hardware and software that provide execution and communication support. Its key constituents are:

- resource sharing (multiplexing) of processing and communication resources;
- a guaranteed real-time service for safety critical applications;
- a best-effort service for non-safety critical applications;
- a communication infrastructure providing efficient communication between distributed devices;
- a program development methodology allowing resource independent and application oriented development of application software;
- a straightforward and well-defined operation principle enabling efficient fault tolerance mechanisms to be employed.

[1] http://www.inria.fr/valorisation/actions-nationales/AEE/RA-2004.fr.html
[2] EAST-EEA project site web: http://www.east-eea.net/

The hardware architecture in a BASEMENT system consists of a set of nodes interconnected with a communication network (Figure 2.7). Each node is a self-contained microcomputer, equipped with a network interface and, possibly, a set of sensors and actuators.

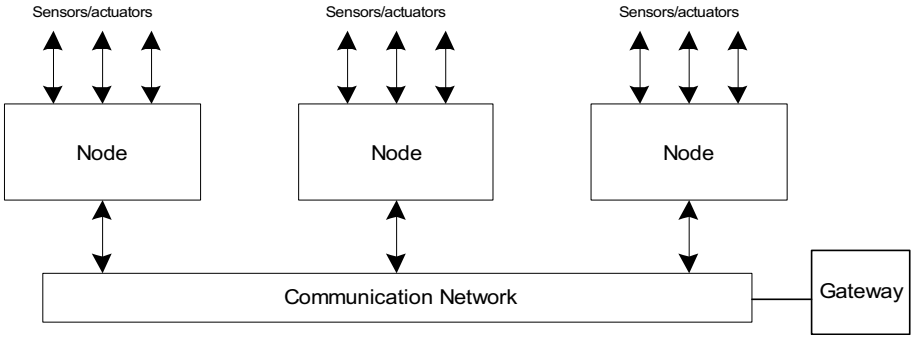



Figure 2.7: Hardware Architecture of a BASEMENT system

The communication network is required to be deterministic, i.e., it should provide error free transmission of data with bounded and predictable delays. The communication network also provides facilities for communication with vehicle external equipment and networks.

The sensors (e.g., the speed of a wheel) and actuators (e.g., set the pressure on the break discs) are used to interact with the outside world.

The scheduling of the critical part of the system is based on a cyclic, static off-line paradigm, guaranteeing timing constraints to be satisfied. Scheduling decisions for the soft real-time part of the system are made at run-time.

BASEMENT is a good example of an embedded in-vehicle software architecture. It reflects the timing and functional requirements of in-vehicle hard and soft real-time applications. Hereafter, we present another solution for automotive application, the COSIMA system, it focuses on the communication between the vehicle and the external environment.

#### 2.4.1.2 COSIMA

COSIMA (Component System Information and Management Architecture) is a platform for services and enabling technologies within vehicles that extends the notion of a distributed system inside the vehicle to the outside world [MMM98].

The hardware architecture of COSIMA system is distributed and consists of two or more nodes. A node may be a computer or any control device. There are two kinds of communication, the *internal* and *external* vehicle communication. External communication allows a reliable communication between the vehicle and the infrastructure. Such services may include Internet access, home banking, etc. As implementation of the *external* communication service, wireless technology is used, this could be GSM (mobile/cellular phones & modems) and wireless LAN technologies.

Concerning the *internal* communication management, the main goal is to deal with telematics communication systems (e.g., MOST, IEEE-1394) and to make devices that are connected to these systems available within COSIMA. "Bridging" between MOST and IEEE-1394 buses is a task that is taken care of by a Protocol Conversion component.

The software architecture of COSIMA is based on a component-oriented model augmented by the idea of completely dynamic behavior of the system. The components can be moved at runtime between

nodes of the system, and even to the infrastructure.

In order to implement these approaches, COSIMA uses a Java technology approach. This keeps the solution open for easy adoption of emerging standards in this field. Moreover, the communication is based on Java RMI but other protocols are also supported using (loadable) proxy components. A test vehicle, equipped with five COSIMA devices, allows to demonstrate the loading of new services. It is connected to infrastructure devices via GSM and Wireless LAN. Finally, streaming data (video and audio) is not supported by COSIMA mechanisms.

### 2.4.2 In-Vehicle network technology

In this section, we introduce present and emerging network technologies in the automotive industry. The network technologies use different types of field buses that meet the requirements of automotive applications and high-level communication protocols. Throughout this section, we will give an overview of automotive network technologies.

There are two main categories of buses for the automotive industry (Table 2.2). The first category is used for the *Engine Control*. This kind of buses is used to control and let communicate highly critical modules when safety is concerned. A typical bus of such category is the CAN bus [Bus93].

| Applications field / Bus categories | Safety and critical applications (e.g., discrete control) | High-level applications (e.g., driving assistance systems) |
|---|---|---|
| CAN, VAN, LIN, etc. | Recommended | Not recommended (low Bandwidth) |
| IEEE-1394, MOST | Not recommended (high safety) | Recommended |

Table 2.2: Main categories of buses

Another category of buses is the infotainment and multimedia buses. Such buses are used in the vehicle to interconnect modules that are not very critical and that demand a certain level of diagnosis and dynamic reconfiguration. Moreover, this kind of buses must support high transfer bandwidth. A typical example of such category is the IEEE-1394 bus [And98] and the MOST bus [Gro00].

#### 2.4.2.1 CAN bus

The dominant bus technology for power train and body electronics in vehicles is the Controller Area Network (CAN) standard [Bus93].

CAN, which was developed by Bosch in the early 1980's and became an international standard (ISO 11898) in 1994, was specially developed for fast serial data exchange between electronic controllers in motor vehicles.

CAN is a broadcast bus designed to operate at speeds of up to 1 Mb/s. Data is transmitted in messages (frames) containing between 0 and 8 bytes of data. An identifier is associated with each message. The identifier serves two purposes: (1) assigning a priority to the message, and (2) enabling receivers to filter messages.

CAN is a collision-detect broadcast bus, which uses deterministic collision resolution to control access to the bus. During arbitration, competing stations are simultaneously putting their identifiers on the bus, one bit at a time (bit-level synchronization). By monitoring the resulting bus value, a station detects if there is a competing higher priority message and stops transmission if this is the case. Because each identifier is sent by only one node, a station transmitting the last bit of the identifier without

detecting a higher priority message must be transmitting the highest priority queued message, and hence can start transmitting the body of the message.

The main disadvantage of the CAN bus is its low data transfer rate, inadequate for infotainment applications and distributed applications used for driving assistance. In addition, the static assignment of identifiers is unsuited for dynamic reconfiguration of systems.

#### 2.4.2.2 Time-triggered buses

Emergent safety critical functions, such as x-by-wire applications, where x may be steer or brake, have forced the development of bus technologies for use in automotive vehicles that meet demands on very high reliability and timeliness. A group of buses that meet this demand have been evaluated by the automotive industry in [Kop00].

These buses are all based on the time-triggered paradigm where the progression of time initiates data transfers rather than asynchronous events. The time-triggered buses provide synchronous communication without the need for arbitration. Moreover they offer mechanisms for redundant networks and have built-in support for a global time base. Therefore the time-triggered protocols are suitable for implementing safety critical control functions with stringent demands on low latency and low jitter. Three time-triggered protocols developed for automotive use are FlexRay, TTP/C, and TTCAN. All these protocols offer services, such as global time and time-triggered communication enabling pre-run-time scheduling of communication. Moreover, these protocols also enable event-triggered traffic to co-exist with time-triggered.

The FlexRay communication protocol [Kop98] supports bandwidth up to 10 Mbit/s with the possible bus topologies, star, and multiple stars. Available communication controllers for the TTP/C [KG94] protocol support 25 Mbit/s for time-triggered transmission and 5 Mbit/s for event-triggered transmission. TTP networks can contain up to 64 nodes and the connection topology can be bus, star, or any combination of the two. Finally, TTCAN [Kop98] is a further development of Extended CAN (version 2.0B), which, like Extended CAN is limited to 1 Mbit/s.

#### 2.4.2.3 IEEE-1394 bus

Since vehicles are becoming equipped with more and more multimedia and telematic applications, the need for dedicated infotainment buses has arisen. The two main buses in this category are the IEEE-1394 bus [NTG02] and the MOST bus [Gro00].

MOST (Media Oriented Systems Transport) is based on optical fiber technology and provides bandwidth up to 20 Mb/s. MOST is very similar to the IEEE-1394 bus regarding the functionalities and services provided. The main difference between MOST and IEEE-1394 buses is the speed. When IEEE-1394 bus reaches 1600 Mb/s (standard), the MOST bus can support 22 Mb/s maximum.

Other infotainment networks relevant to automotive applications are the wireless Bluetooth [JYlM+01] and IEEE 802.11b [SRF03] protocols, used mainly to connect external devices.

Thereafter, we present the IEEE-1394 bus since it is the bus chosen to develop our applications depicted in this work. It is commercially available and affordable.

The IEEE-1394 bus is now suitable for the automotive manufactures to develop in-vehicle networks designed for high-speed multimedia applications that require amounts of information to be moved at high speed within a vehicle.

The IEEE-1394 specification is a high performance serial bus [And98]. This was standardized by

the IEEE in 1995 based on the specification of a bus called FireWire that had been developed by Apple. It can be used to interconnect personal computers, peripheral devices, video decks and digital video cameras. This standard is suitable as a basis for constructing a small-size local area network.

The IEEE-1394 standard has several characteristics that differ from other LAN protocols such as Ethernet. IEEE-1394 specification uses an arbitration method for medium access control. The arbitration method is centralized, and there exists a root node that controls the access to the bus by all nodes in the network. But the root node can change on each bus reconfiguration, so for the transport layer it works as a symmetric bus.

The second difference is that the IEEE-1394 standard specifies two kinds of data transfer modes, namely, *isochronous* data transfer and *asynchronous* data transfer. The IEEE-1394 standard provides *guaranteed bandwidth and latency* for isochronous data and *guaranteed delivery* for asynchronous data.

The isochronous transfer is guaranteed to be *periodic* with bounded jitter (100 $\mu$s). The asynchronous transfer is guaranteed by an acknowledge notification and the worst case transfer delay may be computed using the description of the traffic requests. This feature is useful for the design of real-time systems when guaranteed transactions delivery and bounded transfer delays are required.

In the IEEE-1394 specification, time is divided into fixed-size frames called cycles, each of which has a duration of 125 $\mu$s (Figure 2.8). At most 80% of a cycle is available for transmission of isochronous packets, while the rest of the cycle is available for asynchronous packets. During the asynchronous part of the cycle, any node that wants to transmit an isochronous packet must defer to the nodes transmitting asynchronous packets (the transmission of an isochronous packet has higher priority than the transmission of an asynchronous packet). The transmission capacity for isochronous packets during a cycle is independent of the asynchronous traffic load, but that of transmitting asynchronous packets in a cycle depends on the isochronous traffic load. Because of this asymmetry, the performance of the bus while in asynchronous mode is affected by the traffic conditions in isochronous mode.

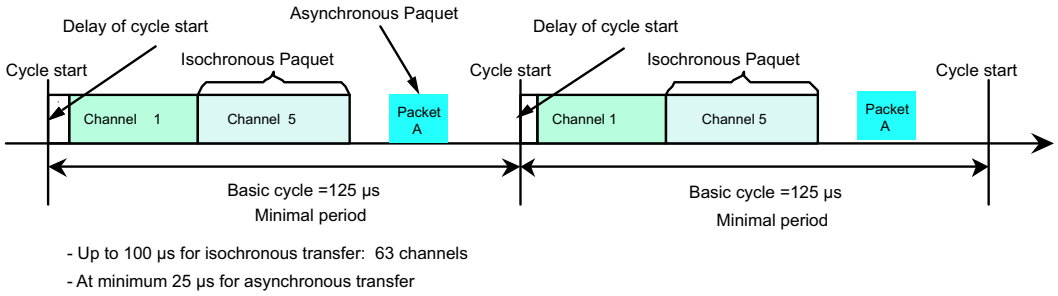



Figure 2.8: IEEE-1394 cycle

An IEEE-1394 network can be reconfigured automatically by its attached devices without any intervention by the users each time the network topology changes. Thus, the system interconnection reconfiguration may be seamless.

On the other hand, IEEE-1394a (version a) does not support the prioritized transmission. All the nodes have the same level of priority. In the next specification of the norm (IEEE-1394b), it is envisaged to have a support for the priority-based transmission, but actually there is no implementation of this specification.

## 2.5 Conclusion

There are a small number of different kinds of middleware that have been developed. These vary in terms of the *programming abstractions* they provide and the software architecture features they supply.

*Programming abstractions* like Remote Procedure Calls (RPC), client/server transactional model, and the Distributed Object Model (DOM) have traditionally simplified and enabled the implementation of complex distributed systems. These programming abstractions served as foundations for successful middleware architectures. Programming abstractions offered by middleware can provide *transparency* with respect to *distribution* in one or more of the following dimensions: location, concurrency, replication, and failures.

Distributed object middleware evolved from RPCs and benefits from all the software engineering advances in object-oriented techniques (encapsulation, inheritance, and polymorphism) available to the distributed application developer. Object middleware provides very powerful component models. It integrates most of the capabilities of transactional or procedural middleware.

Most current middleware are only of limited use in real-time and embedded systems because all requests have the same priority and there is no integrated real-time scheduling strategies. These problems have been addressed by various research groups such as real-time CORBA. Furthermore, distributed real-time systems impose stringent quality of service (QoS) constraints that must be supported by the middleware layer. For example, real-time performance imposes strict constraints upon bandwidth, latency, and dependability.

Recently, the notion of automotive middleware has been introduced by the car industries and third-part suppliers. The main goal is to have a software architecture, shared between car industries and third-part suppliers, ensuring the portability and interoperability of the automotive applications.

The next chapter presents our middleware proposition SCOOT-R developed at our laboratory to design and implement distributed real-time applications.

# *SCOOT-R: Middleware communication services for distributed real-time systems*

This chapter presents our middleware proposition SCOOT-R for *high-level* in-vehicles real-time applications. The middleware is based on a client/server and emitter/receiver models with real-time extensions. Dynamic reconfiguration and fault detection mechanisms are developed to ensure a reliable system. Interactions between processes are performed using the IEEE-1394 bus. Finally, we present a typical automotive application involving our system: "real-time accurate vehicle positioning on a digital map".

## Contents

## 3.1 Introduction

SCOOT-R presents a solution for software development [CCS03c]. It provides a set of basic services built as a middleware layer above a real-time kernel. SCOOT-R offers a framework for distributing components on multi-processing unit architecture, along with communication and synchronization services (Figure 3.1). It also includes run-time monitoring of real-time constraints and ensures a dynamic reconfiguration by replicating software components.

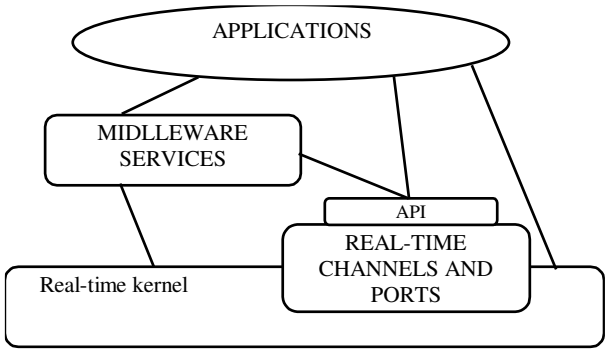



Figure 3.1: SCOOT-R Architecture

Our objective in SCOOT-R was to implement distributed and real-time applications while respecting the applications timing constraints. The system is designed so that it is possible to prove, before launching, that the temporal constraints of applications will be respected. SCOOT6R does not estimate the worst-case execution time of tasks (WCET supposed known *a priori*), but it allows the clients to define their worst case time to have their service (worst service delay). At the server side, the server announces its worst case time necessary to return its service. Thus, before launching, SCOOT-R verifies the compatibility of these timing constraints.

Below, we present the main features that SCOOT-R offers for the developers:

- Communication functions, allowing communication between distributed nodes (based on client/server and emitter/receiver models);
- Dynamic reconfiguration and redundancy approach by software components replication and activation;
- A priori predictability for real-time applications; i.e., it should be possible to determine by static analysis (off-line) if sufficient resources are available to guarantee required behavior;
- A synchronization technique to ensure a global time base for the whole network;
- Simplicity, both in terms of minimal run-time overhead (i.e., minimal amount of nonproductive code), and in terms of intuitive method for application development, which facilitates validation and formal proof of correctness.

All these features described above make SCOOT-R an operational model for the design and development of distributed real-time applications. In addition to these features, it provides some tools and features for the research development in the area of distributed real-time systems. In particular, it provides:

- Resource sharing, i.e., permitting multiple applications to efficiently share communication infrastructure as well as computing resources (processors);
- A framework for the design and development of local/distributed real-time scheduling strategies;
- An application development environment and methodology, providing researchers with an application oriented interface, as well as tools for efficient development and integration of applications.

Our focus in this work has been on how to design a working system satisfying the above listed requirements, rather than solving a very specific technical problem.

While SCOOT-R was designed to be used mostly in control/command distributed systems, the main target application depicted in this thesis is the automotive applications. i.e., SCOOT-R offers an integrated framework to design and evaluate driving assistance functions. In particular, it allows to acquire embedded sensors data at run-time and then to perform on-line computation. A set of driving metrics and indicators was defined to implement several driving assistance functions. The use of SCOOT-R in such applications is justified by the need of distributed, modular and flexible architecture that provide diagnostic and synchronization services. The next section presents the context of our research activities depicted in this thesis.

## 3.2 Research context

Given the research activities of our laboratory in the field of the "advanced vehicles" and "data fusion" [NB05][PG96][PG98][VB94], and our work in the "real-time" field [CCS03c][CCS03b][CCS03a][CSC04], the framework of our research activities in this thesis was mainly the European project RoadSense (ROad Awareness for Driving via a Strategy that Evaluates Numerous Systems), the national research project ARCOS (Action de Recherche pour une COnduite Sécurisée), and Mobivip (Véhicules Individuels Publics pour la Mobilité en centre ville) national project.

This thesis was supported by the European project RoadSense whose main objective is to provide automotive manufacturers by analysis tools in order to enable them to define the guidelines to evaluate the relevance of the Advanced Driving Assistance Systems (ADAS) in next generations vehicles.

The HVI (Human Vehicle Interface) is the main point studied by this project. A certain number of criteria was defined describing the reaction of the driver regarding the ADAS system. The driver performances are thus evaluated, with and without the ADAS system, in order to assess its contribution to the improvement of safety.

The objective is thus to provide the processing tools necessary to the calculation of these criteria, which will be called in this document human factor metrics or indicators.

We have contributed to the development of a design methodology, realization and deployment of a distributed real-time system. The system, called D-BITE, based on SCOOT-R middleware, with respect to the study case chosen by Renault.

D-BITE system allows the perception and processing of large amount of data. These data are acquired from the physical sensors and digital cameras embedded in the vehicle. In order to provide the relevant assistance functions, we have defined a set of driving metrics and indicators. These indicators are computed in run-time during experimentation courses and they are recorded on several embedded hard disks aboard the vehicle.

As a simple example of these metrics, we can cite the "Exceed speed detection". To compute this indicator, we need to compare the current vehicle speed with the maximum allowed speed on the

current road. The calculation of this metric involves the ABS data to get the current vehicle speed, the GPS data to get the current position and the GIS (Geographical Information System) data to map the current vehicle position on the digital map and extract the maximum vehicle speed allowed on the specified road.

Another example of metrics is the "actions on pedal". It is the number of pedal (accelerator and brake) depressions, mean throttle position and position variance. The actions are considered versus time (mn) or distance (km). For Renault case study, only the actuations had been considered for the analysis; the acceleration is got from the CAN bus. The throttle position and the position variance of the accelerator are calculated *a posteriori*. Moreover, a camera is pointing toward the feet of the driver to see the actuations intentions on the accelerator and braking pedals.

The use of a middleware technology like SCOOT-R made the implementation of the component architecture of D-BITE simpler, since SCOOT-R manages the real-time execution of all the components (starting, monitoring and stopping) and their synchronisation and communication.

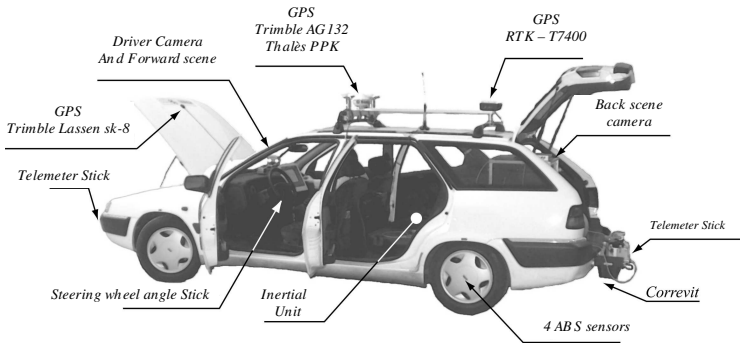



Figure 3.2: Sensors of the STRADA vehicle

In order to test and evaluate our D-BITE system, we used our demonstrator car STRADA (Figure 3.2). STRADA is equipped with certain number of sensors (Figure 3.2). This includes the GPS (Global Positioning System), the odometers (one per wheel), the accelerometers, the driver commands sensors (eyes-tracker, brakes, accelerator, etc.). Embedded digital cameras (IEEE-1394 cameras) were added to record the face of the driver, the external scene of the vehicle (forward and back scenes) and the driver foot position on the accelerator pedal to detect its intentions. All these data are collected during a course of experimentation (twenty experimentations, two hours each), and are recorded and time stamped on several embedded hard disks aboard the vehicle.

The system specifications must be sufficiently flexible to allow several configurations, according to the case study chosen by the automotive manufacturer. The case study is defined based on the ADAS system to test. Several modules were developed in order to replay the experiments recorded data with search possibility by simple criteria or sequences indexing [MSP00].

D-BITE is a set of equipments that can be embedded in the car equipped with the ADAS functions or located in a room for the Human Factor analysis (Figure 3.3).

Another important research project that involves our research works depicted in this thesis is the French project ARCOS. ARCOS is a pre-competitive research project, it takes place into the PREDIT program. This project aims at improving road safety and considers vehicle, driver and road as a whole system. Thus, the project aims at enhancing driving safety on the basis of four safety functions:

- Controlling inter-vehicle distances;

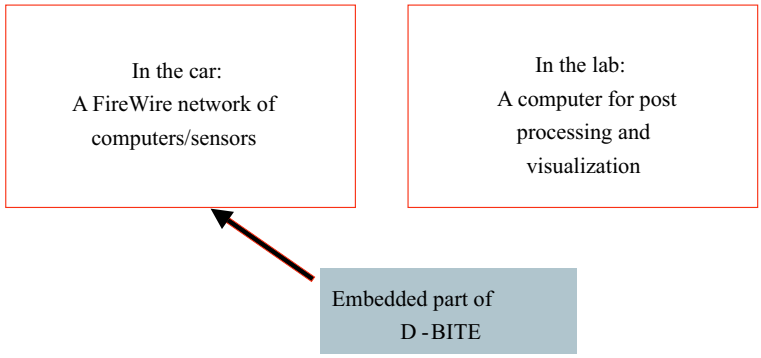



Figure 3.3: The two instantiations of D-BITE

- Avoiding collisions with fixed or slowly moving obstacles;
- Avoiding lane exit;
- Alerting other vehicles of accidents by vehicle to vehicle communication.

Let's recall that all the perception and processing functions are encapsulated into software components using our middleware proposition SCOOT-R. Hereafter, we present the features of SCOOT-R that make of it a good solution for prototyping multi-sensor automotive applications. We begin by the description of the hardware (§3.3) and software architecture (§3.4) of SCOOT-R. Then, we present the dynamic reconfiguration (§3.5) and the services operation (§3.6). Finally we outline some performance keys (3.8) of SCOOT-R and a typical automotive application involving it (§3.9).

## 3.3 SCOOT-R hardware architecture

A SCOOT-R system consists of a set of nodes interconnected by a communication network. A node can be viewed as a computer (processor + main memory) with a network interface and a set of input/ output devices (sensors and actuators) allowing interactions with the "physical process" (e.g., the vehicle) (Figure 3.4). The communication network has to be deterministic, i.e., it should provide data transmission with bounded and predictable delays and it should provide also a synchronisation mechanism to define a global time base.

As the main goal of a middleware is to mask the heterogeneity of computer architectures, operating systems, programming languages, and networking technologies, our SCOOT-R middleware supports multi-platforms. It interfaces heterogeneous nodes with heterogeneous operating system. A node in our system can be:

- a PC (Real-Time Linux or Windows OS);
- a microcontroller: we are currently working on the development of an embedded version of SCOOT-R for the PowerPc controllers (MPC555) and the Embedded OS OSEK/VDX.

Furthermore, a CAN gateway application was developed in our laboratory to ensure the communication between SCOOT-R applications and the raw CAN data.

As communication media, SCOOT-R uses the IEEE-1394 bus. The IEEE-1394 network has a common clock so that devices can synchronize their operations; this is suitable for applications and

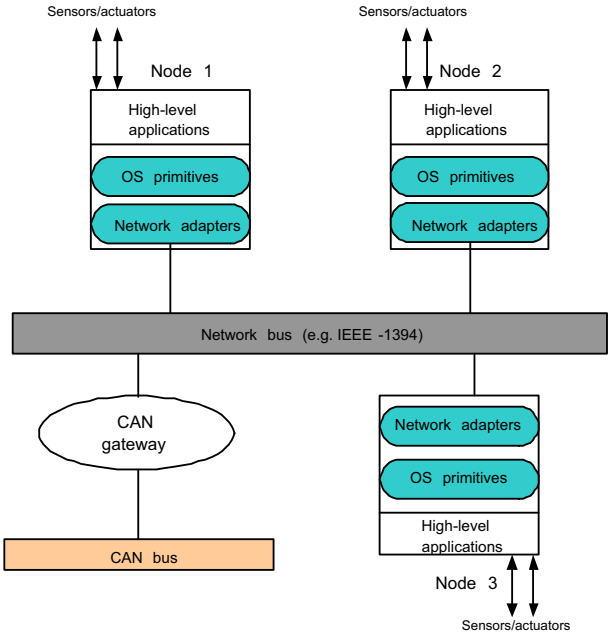



Figure 3.4: SCOOT-R Hardware Architecture

also for the diagnosis and dynamic reconfiguration techniques. The choice of the IEEE-1394 bus as a communication media was mainly imposed by the partners of the RoadSense project. In addition, the IEEE-1394 bus has also a set of interesting features that make of it a good candidate to implement distributed real-time systems (e.g., high bandwidth, dual transfer modes, synchronsiation, and dynamic network management). The IEEE-1394 bus was briefly presented in section 2.4.2.3.

## 3.4 SCOOT-R software architecture

In SCOOT-R, the communication and synchronisation between application components follow either the *client/server* model, or the *emitter/receiver* one. As outlined in Figure 3.5, SCOOT-R middleware services are sets of distributed software that exist between the application and the operating system on a system node in the network.

Each application component is located on one node in the distributed system. The client/server model as the emitter/receiver one are extended to allow explicit real-time *protocol specifications* (§3.4.2 and §3.4.3).

SCOOT-R is based on object-oriented programming and allows remote method invocations on objects. It supports objects that are data abstractions with an interface of named operations and a hidden local state. Objects have an associated type [class] and the types may inherit attributes from supertypes.

SCOOT-R communication layers cover from network to presentation layers and lie on the physical layer of the used network (IEEE-1394). The presentation layer encapsulates the full set of communication features in an object oriented API (Figure 3.6).

Currently, SCOOT-R beta version is running under the microkernel RTAI (§1.3.4). Using RTAI, the system may turn in a hard or soft real-time mode. Thus, we have two modes of service implementation: "Kernel mode" and "LXRT mode". In kernel mode (Figure 3.7(b)), the servers (emitters) and clients

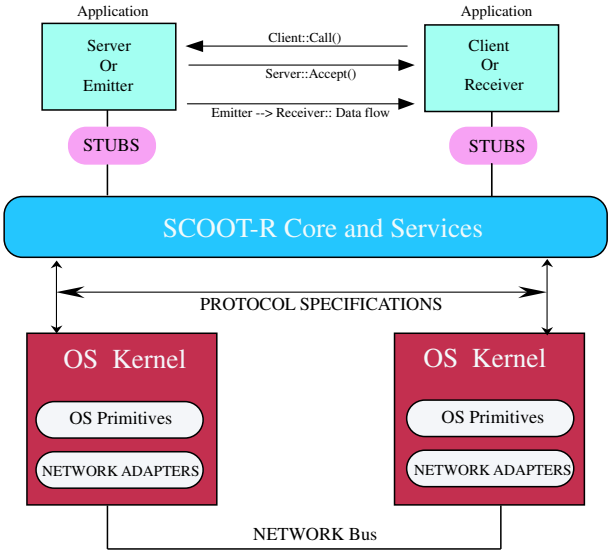



Figure 3.5: Components in the SCOOT-R Model

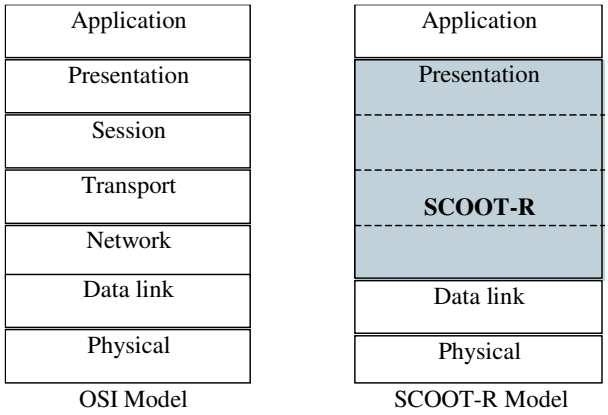



Figure 3.6: SCOOT-R vs. OSI Model

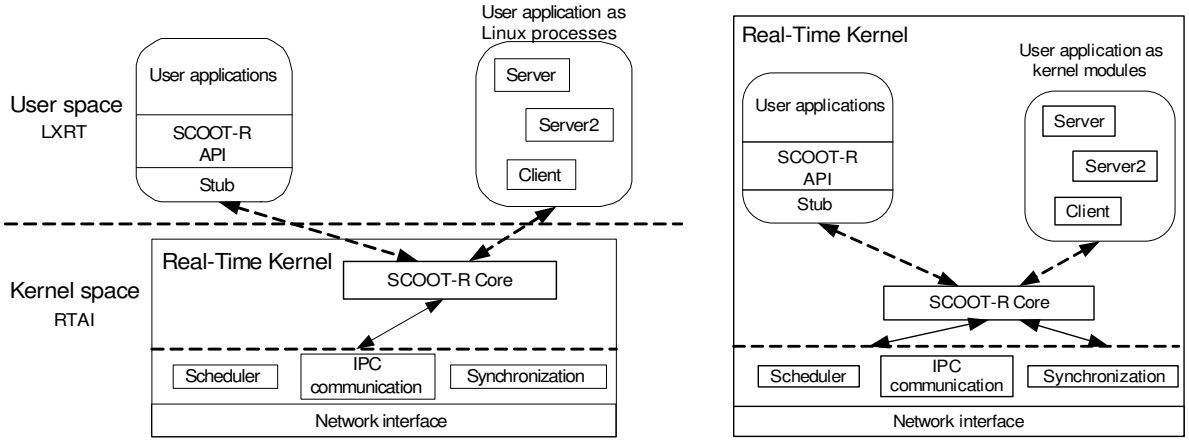
(receivers) are implemented as kernel modules in the kernel space. In the "LXRT mode" (Figure 3.7(a)), the services are implemented as Linux standard processes.

Another version of SCOOT-R was also developed for Windows platforms (Windows 2000, XP). This implementation is not a real-time version, and it is mainly used by the developers that aim to access Windows software components such as the GIS (Geographical Information System) and to communicate with the RTAI platform to acquire and process real-time data.

**Modular and component-oriented architecture** is an important characteristic of SCOOT-R. It is now recognized that object-oriented techniques are well suited to the design and implementation of distributed real-time applications [Kop97a]. Objects may be used to encapsulate a great variety of hardware devices used in such applications and to make abstraction of the low-level interface details.

The object-oriented concept has been widely used in SCOOT-R: each element of the system is an object. Thus, one can easily write his own custom object in C/C++, and SCOOT-R provides the interface to the other objects (e.g., sensors, display systems).

In our philosophy, a real-time system is constituted by interconnected software components (Figures 3.8 and 3.9). Each of them contains a set of servers, emitters, clients and receivers, and possibly other




(a) User space implementation (LXRT mode)

(b) Kernel space implementation (Kernel mode)

Figure 3.7: Service implementation

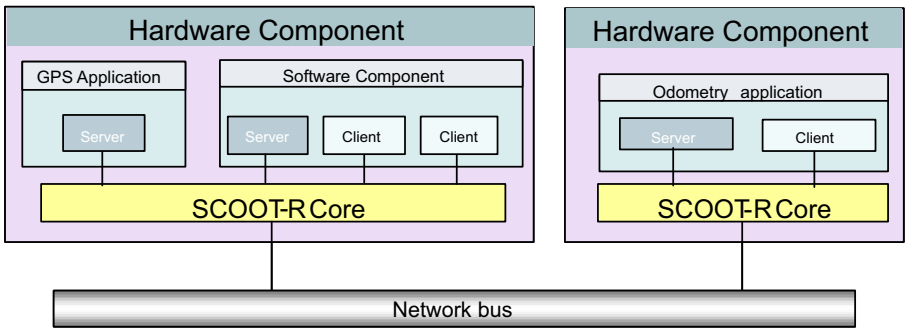



Figure 3.8: Component-oriented architecture – client/server model

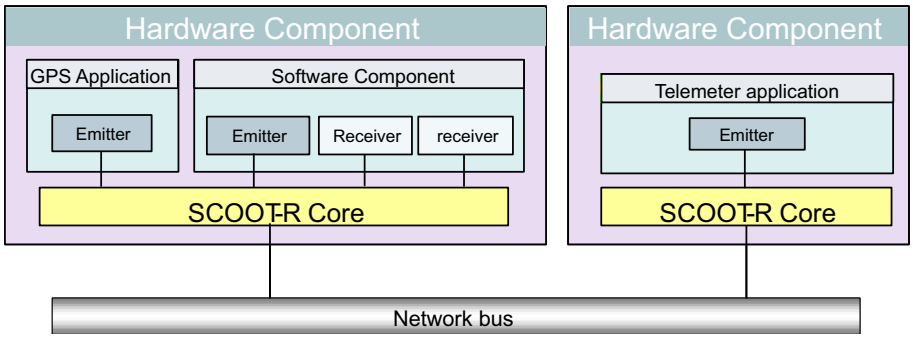



Figure 3.9: Component-oriented architecture – emitter/receiver Model

tasks that cooperate between them to provide the results expected by the component interfaces.

A component is a self-contained function that can be used as a building block in the design of a larger system. The component provides the specified service to its environment across the specified component interface. The component can have a complex internal structure that is not visible to the user of the component.

The development of standard real-time components that can be run on different hardware plat-

forms is complex. The components have different timing characteristics on different platforms. Thus a component must be adapted and re-verified for each HW-platform to which it is ported, especially in safety critical systems. Hence, we need to perform a timing analysis for each platform to which the system is ported.

### 3.4.1 Failure detection and recovery in SCOOT-R

The SCOOT-R model for failure detection and recovery supports several schemes and situations of failures. Failures may be *hardware* or *software* and they may be *permanent* or *transient*. SCOOT-R processes the eventual failures by the release of watchdogs. This processing consists in a reconfiguration limited to the concerned failed service (*partial reconfiguration*) but following the same schema used by *the global reconfiguration*. The failures that may lead to a watchdog release are:

- Component failure: a crash of a software component due to a software error has been occurred (e.g., internal inter-blockage). For SCOOT-R, the server that resides on the failed component still exists, but the execution of the client request is not terminated. Thus, the client detects a "timeout" error and considers the server as failed. This *timing* failure may be classified as *permanent* error and it leads usually to a *partial reconfiguration* for the concerned service.

- Breakdown of a node: the whole node shuts down. This error may be due to an OS crash (e.g., RTAI microkernel) or to a crash of the SCOOT-R software layer, or due to an error on the network interface (IEEE-1394) which leads to the non transmission of data. This failure is *permanent* and may be classified as *fail-silent* failure. In this case, the clients that try to perform transactions with a server located on this failed node will miss their transactions by a release of their watchdogs. Partial reconfigurations will be performed respectively for each server located on this failed node.

- Transmission error on the IEEE-1394 network bus: the client request or the reply of the server has been lost or arrived but incorrect (the retransmission mechanism of the IEEE-1394 bus is deactivated, if not, it is difficult to compute the emission time on the bus). This kind of failures is *transient*, it does not occur permanently in a nominal operation of the system and it leads to a partial reconfiguration.

- Network overload: this overload may lead to longer than specified communication delays; a complete study and a conform deployment of the system components may permit to prevent this kind of situations. However, a design error or a non-controlled connection of a node or application consuming high bandwidth may lead also to such situations. In this case, "timeout" can occur (transaction failed) and that implies *partial reconfiguration*.

- Reset: a reset on the network may be initiated by several reasons: incidents detected by the IEEE-1394 network adapter (hardware reset) that may be resolved by the reconfiguration of the network. Another kind of reset is the *software reset* initiated by the SCOOT-R middleware layer in case of adding of a new node (starting of the SCOOT-R software stack). A reset leads generally to a *global reconfiguration* of the system.

On the other hand, there is no detection mechanism and processing of *values failures* (e.g., the response arrives completely but incorrect). Furthermore, there is no support in SCOOT-T to restart or reinitialize software or hardware failed components and to manage redundant network buses.

### 3.4.2 Client/server communication model

The client/server model has proved its efficiency for the development of network applications. Clients and servers can be implemented by independent programmers teams. Once the communication protocol is specified, the client and server code may be distributed.

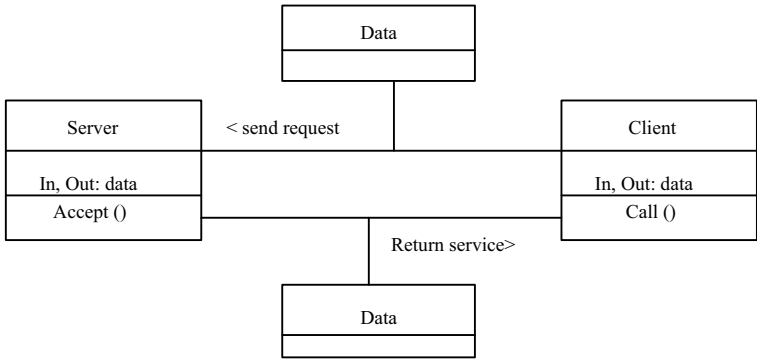



Figure 3.10: Object oriented architecture – client/server Model

Moreover, the use of a client/server model facilitates the object modeling of the system. As depicted in Figure 3.10, each client and server of the system is modeled by an object. The interactions between the applications are materialized by invocations of objects' methods. SCOOT-R implements a very simplified scheme: each server has only one remotely accessible method ("accept()" method).

SCOOT-R clients and servers exchange data while respecting a real-time contract, which is specified as a set of standardized constraints. Table 3.1 illustrates these temporal clauses and the results in case of non-respect of the rules. The runtime monitoring of the deadlines by watchdogs allows immediate detection of failure. i.e., the watchdogs are associated to system exceptions. Thus, the expiration of a "timeout" will launch the exception that will be examined by the SCOOT-R software layer.

Let's note that the attributes in Table 3.1 are individual and depend on each client and server of the system. Moreover, SCOOT-R client/server paradigm is an event-based system and is implemented using the asynchronous transfer mode of the IEEE-1394 bus. The SCOOT-R data object following a client/server model is encapsulated in an asynchronous IEEE-1394 packet. The maximum size of an asynchronous packet is bounded (1024 bytes for the SCOOT-R Windows interface SEDNET) and thus, the size of a client/server data object is limited. Generally, this data representation is sufficient for a wide spectrum of applications. In case of non compatibility with the application requirements in term of message size, the programmers can define several servers and then associate to each server a part of the request data and thus decreasing the data request size. Another solution is to encapsulate in the SCOOT-R layer an assembling/disassembling module that permits to send a data request on several consecutive asynchronous packets.

### 3.4.3 Emitter/receiver communication model

Real-time emitter/receiver is a "classic" model to which we add timing constraints as contracts between the emitter and the receiver. The essential point in such system is that the receiver consumes data objects produced by the emitter at the same rate they are delivered. Moreover, real-time emitter/receiver is a broadcast-based protocol without registration of consumers close to producers. This paradigm is particularly adapted to image acquisition and analog signal acquisition.

| | CLIENT SIDE | SERVER SIDE |
|---|---|---|
| Constraint attribute | Description | Description |
| Recovery time on simple error | This clause is an engagement of the server and a request of the client | Maximum time to register a server in the case of desactivation of the server or some incidents such as a reset on the IEEE-1394 bus |
| Min_Period | Maximum rate of request to be respected by the client. It must be compatible with the server constraint | Maximum rate of requests for each client. This value multiplied with the maximum number of clients limits the processor resources needed by the server |
| Max_clients | N/A | Maximum number of simultaneous clients supported by the server |
| Worst_Case | This time includes the response time and the communication time in the worst case. If it expires, the transaction fails with "time-out" error. The server is considered defective and its client may replace it by a redundant server | Response time of the server in worst case. The non-respect of this time indicates a software misconception rather than hardware or network error |

Table 3.1: Real-time contracts: client/server model

As depicted in Figure 3.11, each emitter and receiver of the system is modeled by an object. The interactions between the applications are materialized by invocations of objects' methods.

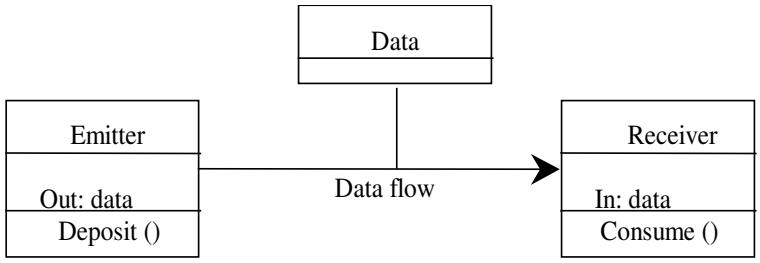



Figure 3.11: Object oriented architecture - emitter/receiver model

Currently, there is a strong trend in the vehicle industry to increase the use of telematic and infotainment services. Examples include fleet management systems, maintenance systems, and anti-theft systems. Consequently, a data-flow driven model represents an important solution for such in-vehicle applications. An emitter/receiver model may be used to fill these requirements. As the client/server model, the emitter/receiver model permits the software component development and is well adapted for the distributed computing.

Emitters and receivers in our model exchange data while respecting a real-time contract which is specified as a set of standardized constraints. Table 3.2 illustrates these temporal clauses and the results in case of non-respect of the rule.

Using the IEEE-1394 bus, the emitter/receiver paradigm is easily implemented using the isochronous transfer mode. The basic cycle is the IEEE-1394 cycle of 125 $\mu$s and the number of channels is limited to 63 (we have one emitter per IEEE-1394 channel).

| | RECEIVER SIDE | EMITTER SIDE |
|---|---|---|
| Constraint attribute | Description | Description |
| Recovery time on simple error | This clause is an engagement of the emitter and a request of the receiver | Maximum time of registering of the emitter in the case of desactivation of the emitter or some incidents such as a reset on the IEEE-1394 bus |
| Nominal_Period | N/A | Central value of the emitter period |
| Min_Period | Maximum rate of data stream. It must satisfy: Min_Period $\leq$ Nominal_Period - Jitter | N/A |
| Max_Period | Minimum rate of data stream. It must satisfy:Max_Period $\geq$ Nominal_Period + Jitter | N/A |
| Jitter | N/A | Dynamic adjustment of the emitter period |

Table 3.2: Real-time contracts: emitter/receiver model

At the emission side, a data object (the image for example) is broken down into portions of identical size so that each portion is sent in an isochronous packet of the IEEE-1394 cycle. These portions are chained between them forming the whole data object. At the reception side, the receiver performs an assembling method to rebuild the whole data object.

### 3.4.4 Client/server invocations

In order to explain the philosophy of our model operation, Figure 3.12 and Figure 3.13 illustrate two simplified statecharts of our system for the *client/server communication model*. These statecharts provide a simple illustration of the system operation by presenting the essential states of the system.

Below, we define the *quality indicator* used by SCOOT-R software layer to choose the best service in case of redundant software components for the same service.

**Definition 4.** ***SCOOT-R quality indicator:*** *The quality indicator associated to each service (server or emitter) is a positive integer that may be zero. It expresses the quality of data (e.g., accuracy, precision and confidence level), depending for example of the condition of the external environment (e.g., vehicle visibility and temperature). This indicator is used by the middleware layer to select the best service in case of redundant components. The high-level applications are allowed to modify this quality value dynamically at runtime. This change will be taken into account by the middleware layer by informing all the nodes on the network.*

The "remote server" object (client side network interface) is a local representative of the server. As shown in Figure 3.12 and on the client node, the client application starts the transaction by invoking the *call()* method of the "remote server" object. This call is invoked sporadically by the client application ("next client invocation" loop in Figure 3.12).

The "server status" represents the current status of the server associated to the client as stored in the local SCOOT-R services table. It has two steady states: the state "on" and the state "off" that correspond respectively to a *registered* or *unregistered* server state.

When the client application initiates a client transaction by the *call()* method, normally the server responds to the client transaction by the *accept()* method if it is in the "on" state (registered). If the

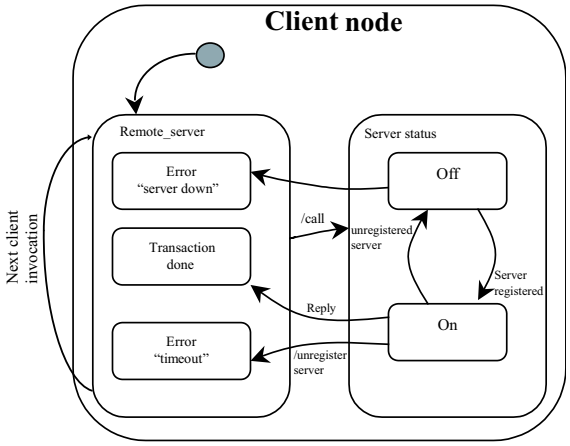



Figure 3.12: Simplified statechart of a client operation

server is registered but it does not respect the client expiration time (timeout) of the client transaction, the server is considered by the client as defective and it will be unregistered by the client. However, if the server state is "off", no communication is initiated and the local services table directly return to the "server down" state.

At the **server side**, initially the server is in the "off" state. It leaves this state and can be considered as soon as its quality becomes greater than zero (Figure 3.13). In this case, the server will switch to the "ready" block and it is considered as *registrable*. It will be registered if it has the highest quality of all others servers of the network for the same service ("Better" state). If not, it remains in the "Degraded" state.

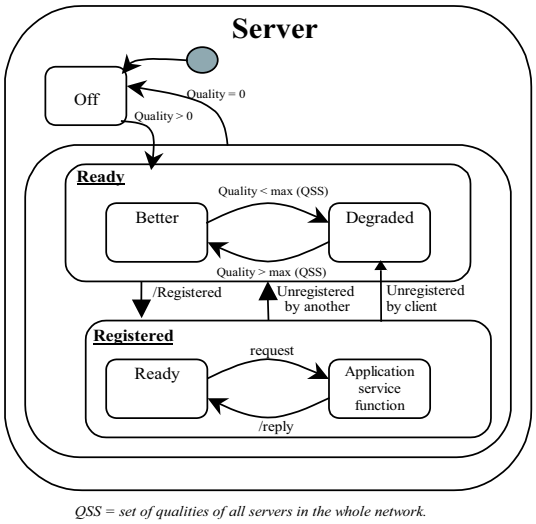



Figure 3.13: Simplified statechart of a server operation

Now, the server is registered (it has the highest quality), it will pass to the "registered" block and will operate normally by responding to the client invocations. The server leaves the "registered" block in case of unregistering of it.

The server may be unregistered by (1) the client (in case of "timeout error"), (2) by a *bus reset* that occurs in the network and introduces a global reconfiguration or (3) *another server registration* that

leads to the re-registration of all the services available on the network, or (4) because *server quality becomes zero* that occurs when the server application decides to stop it.

When the server is unregistered by a client detecting a "timeout", it goes to the "degraded" state, its quality is set to "1", the smaller non "0" quality value, so it can be registered only if there is no another server for this service.

In case of normal operation of the client/server model (Figure 3.14 (a)), the client initiates the transaction by invoking the "call" method and a watchdog is activated for each transaction. The server responds to the request by the "accept()" method. The response must be returned in a bounded time (as announced in the real-time contract terms, see Table 3.1). If it is the case, the watchdog is reset to the "timeout" value mentioned in the client real-time contract.

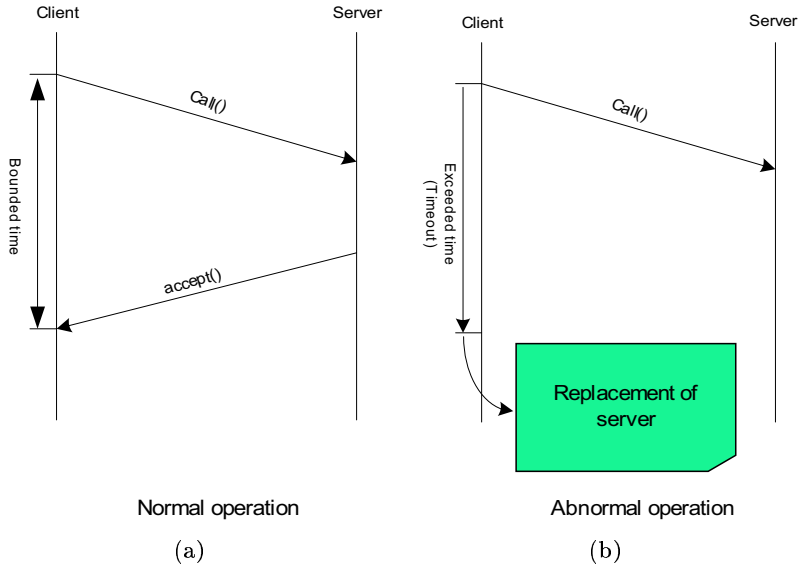



Figure 3.14: Client/server operation modes

The exceptions that may occur at run-time are:

- server_down: there is no server registered for this service at the time of the request. This happens in initial phase or when the server is failed.
- timeout: the server exists, it should render the service within the bounded time, but it does not. The cause of the problem can be a network failure, a hardware failure where the server is located, or a failure of the software on the server application itself. In all cases, SCOOT-R software revokes the server, which enables another possible server rendering the same service to take its place and allow the resumption of the service. This failure case is illustrated in Figure 3.14 (b) and cited in Table 3.1;
- overrate: the client does not respect the requests frequency bounds announced by its server, the service is not rendered. This exception may occur when there is a system clock problem on the client node.

In case of the two exceptions "server_down" and "timeout", the system reacts by a partial reconfiguration of the associated service. For the "overrate" exception, there is no recovery and handling of the error.

Other exceptions may occur in the system and they are used specifically to monitor and diagnose the system. For these exceptions, we do not have any special treatment and recovery. These exceptions are:

- type mismatch: the object types exchanged between the client and the server are not compatible. The client will not be able to communicate with the server.
- too many clients: the server cannot support this client (maximum simultaneous clients number has been reached). If it accepts this client, it could not guarantee the respect of its temporal requirements (see Table 3.1);
- timeout too short: the client asks for a response time lower than the one announced by the server. The service cannot be rendered.
- rate too high: the client announces a requests frequency higher than the maximum value supported by the server. The server could not respect its temporal engagements if it accepts this flow of requests.

The verification of real-time contract clauses is made at pre-runtime for some clauses and at run-time for others. For example, the "type mismatch", "too many clients", "timeout too short" and "rate too high" exceptions are detected before the effective execution of communication between clients and servers. The "overrate" exception is handled at run-time by the client side since it occurs when the client overcomes the initially announced frequency.

As each transaction consumes some CPU time, the total server activity is calculated using "Min_Period" and "Max_Clients". The load distribution on the processing units is thus validated (Figure 3.15).

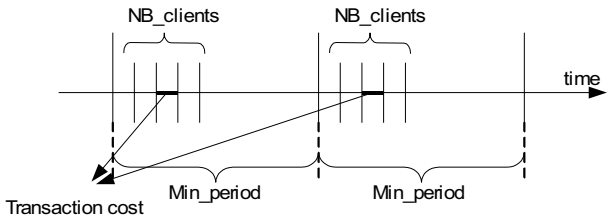



Figure 3.15: Transactions chronogram

In a real-time application including several clients and servers, the processor load induced by this application can be evaluated using the traditional techniques (e.g., RMA analysis).

Given that the number of simultaneous clients per server and the requests frequency of clients are bounded, the worst case CPU load is known. This analysis enables to validate both the respect of the temporal clauses of the server and the operation of the real-time application.

#### 3.4.4.1 Emitter/receiver invocations

As depicted in the Figure 3.16, a data object is sent using one or more IEEE-1394 isochronous packets. During an IEEE-1394 cycle, one packet of the data object is sent on the bus (P1, P2, ..., Pi).

The emitter sends its data objects at a specified *period*. The receiver checks if the period of the received data objects is comprised in the interval [period - jitter, period + jitter].

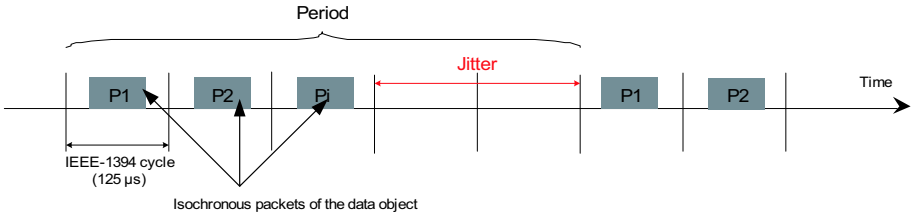



Figure 3.16: Jitter

The Figure 3.17 illustrates two system behavior of the emitter/receiver model: operational and failure modes.

In normal operation (Figure 3.17 (a)), the emitter broadcasts its objects on the network (by calling the "deposit()" method) and the receiver(s) consume these objects at a frequency compatible with the one announced by the emitter (see Table 3.2).

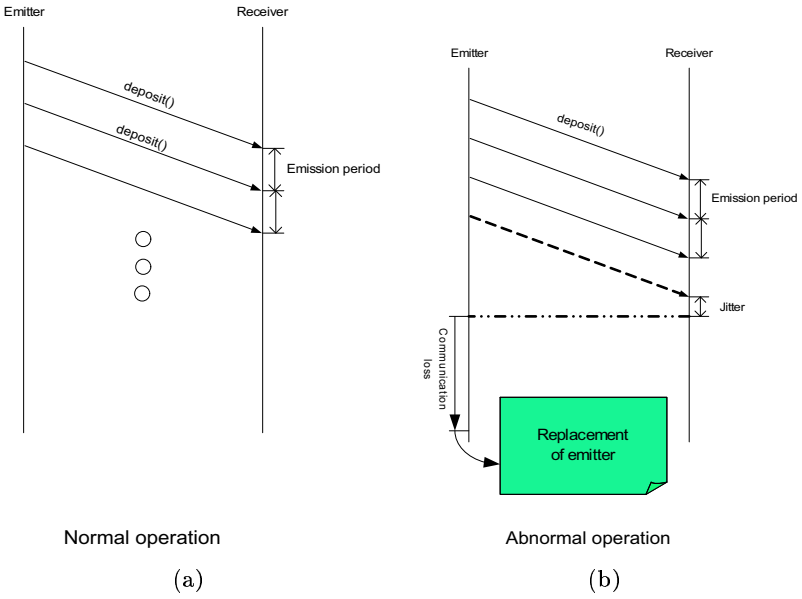



Figure 3.17: Emitter/receiver operation modes

In case of communication loss or any other incident on the network, the receiver detects the failure using a timeout watchdog and then it considers that the emitter has failed. It initiates the unregistration of the emitter by decreasing its quality and thus it opens the possibility of an eventual replacement (Figure 3.17 (b)).

SCOOT-R kernel uses its services table to check if there is another active emitter for this service, if so, the new emitter will be registered in a bounded time and the receiver will continue to consume data from this new emitter. If not (there is no active emitter for the service that failed), the emitter returns an exception (no active emitter for the specified service). The registration algorithm is detailed later in this chapter (see section 3.6.3).

During the start up of an emitter, there may be a problem of synchronisation between the emitter and its receiver(s). In this case, the receiver may receive an incomplete object. Thus, the receiver must tolerate the isolated missed objects. i.e., the receiver should wait a sufficient time before the

unregistration of the emitter by considering it as defective.

## 3.5 Dynamic reconfiguration and redundancy management

SCOOT-R uses a partial replication of the software components. The mechanism used to detect and recover from failures is based on components redundancy. In the fault detection phase, the error state and the needed reconfiguration are identified.

In fact, SCOOT-R uses the server and the emitter redundancy to implement services replication. A specific algorithm, described in section 3.6.3, will permit the activation of the better server registered in the services database replicated in each node of the network.

Let's recall that the redundancy approach in our system is initialized by the programmers, i.e., the high-level application must define, declare and then implement the redundant services. Then SCOOT-R ensures the error detection and the activation of the available service having the best quality factor.

In order to respect the application's timing constraints, the restoration time of the service should be compatible with the temporal constraints announced by the application. The registered server waits for input requests from the client and sends a response to the client to mark the end of an invocation. In addition, all other servers replicas (not registered) are in a "standby" mode. i.e., these replicas do not receive clients requests and they wait to be registered in case of an eventual reconfiguration operation.

In case of adding a new node on the bus, it should be informed of all the previously registered services of the other nodes. For IEEE-1394 bus [And98] or for MOST bus [Gro00], a global reconfiguration is made whose over-cost may be bounded. This global reconfiguration is also used for the node suppression or other major network incidents.

Moreover, there are three main schemes that may lead to a reconfiguration operation and consequently to register and replace a server (emitter):

- global reconfiguration: all the active services registered on the network are forgotten, each server previously registered or not initiates a procedure of re-registration by broadcasting its registration message to all nodes. Using the IEEE-1394 bus, this happens each time a bus reset (soft or hard) occurs;
- timing constraint violation and server shutdown by the client (detection as watchdog "timeout"): this case occurs when the server does not respond on time and the client has got a *timeout* exception. So, the client has the capacity to consider the server as failed and shut it down. To be reactivated, the server must initiate a registration procedure to introduce itself again. At the emitter/receiver side, we have the same behavior, i.e., the receiver detects a loss of communication and it considers that the emitter has failed. The receiver has the capacity to shutdown the emitter and a mechanism of replacement may occur;
- quality indicator change: introduction of a new server with a higher quality than the current registered server or the decreasing of the current active server's quality factor. In this scheme, there is no service interruption.

Let's consider an accurate positioning system of a vehicle functioning by fusion of inertial and GPS data. Another server yields position using GPS and Odometer data (Figure 3.18). A quality indicator is associated to each server.

This quality may depend on the vehicle physical environment as speed, adherence, etc. Each fusion module evaluates its quality. When the positioning data is needed, SCOOT-R provides the value having

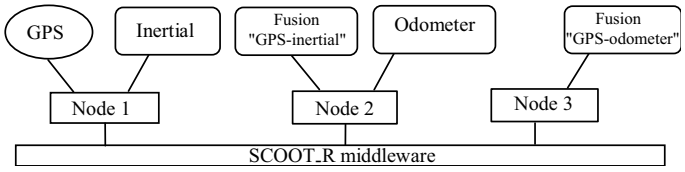



Figure 3.18: Redundancy approach

the best quality indicator. Let's assume that "GPS-inertial fusion" module will be providing the best value. In case of "Inertial" module failure, the quality indicator of the "GPS-inertial fusion" module decreases significantly. Hence, SCOOT-R selects dynamically the "GPS-odometer fusion" module.

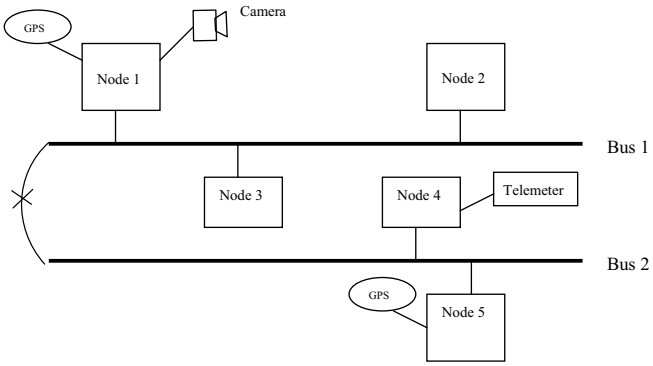



Figure 3.19: Redundancy of servers on duplicated buses

Moreover and as illustrated in Figure 3.19, a service may be duplicated on several separated buses of the whole network. When these buses are merged, the Bus 1 and Bus 2), the two instances of the same service (in our case, the GPS service) candidate to be operational and SCOOT-R software layer selects the service that have the best quality.

## 3.6 SCOOT-R internal services operation

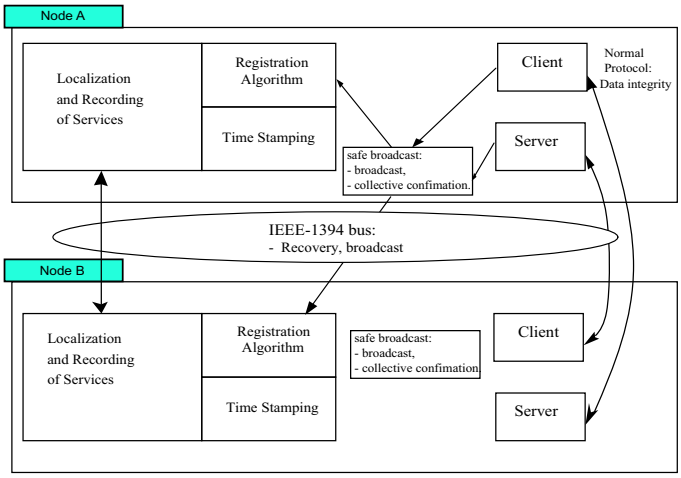



Figure 3.20: SCOOT-R service operation

SCOOT-R needs a communication network providing a deterministic media access time and a bounded transmission delay in order to provide the real-time services presented hereafter.

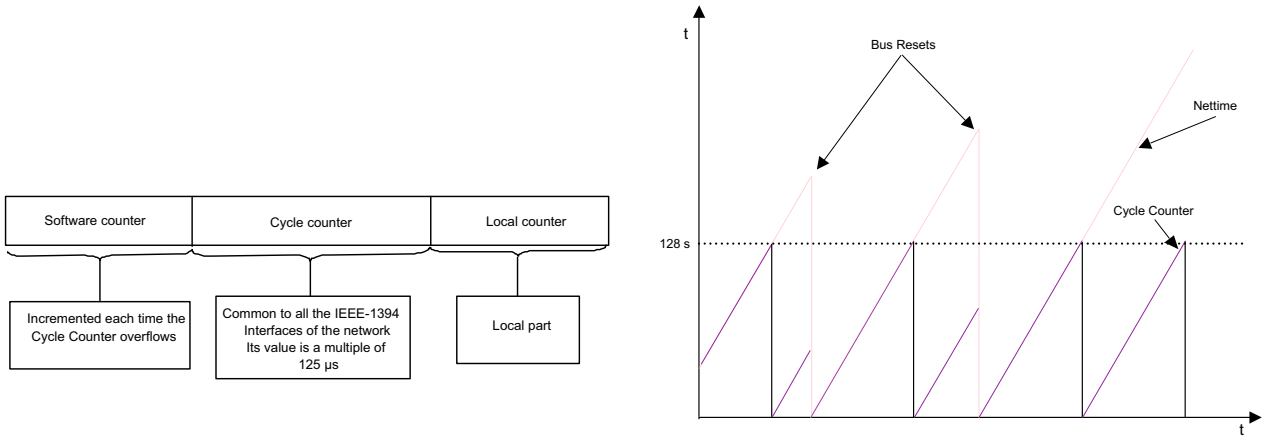
As depicted in Figure 3.20, SCOOT-R invokes several modules that cooperate in order to guarantee correct operation of the system. Among these modules, we mention the "Time Stamping", "Localization and Recording of Services", "Safe Broadcast protocol", and "Decision Algorithm". These modules are detailed in next subsections.

### 3.6.1 Time stamping

In order to guarantee a consistent behavior of SCOOT-R, it must be ensured that all nodes process all events in the same consistent order. For all the nodes on the network, the events chronology must be the same. A global time base helps to establish such a consistent temporal order on the basis of the timestamps of the events.

The IEEE-1394 bus offers a service of high rate synchronisation (8 kHz) that permits to obtain a global time between two bus reconfigurations. A reconfiguration occurs each time a node is added or removed.

We note "nettime" as the network time used by the nodes to compute the global time base. It is built using the Cycle Counter register of the IEEE-1394 bus. Each 128 seconds, the Cycle Counter register is reset to zero and the overflow of the Cycle Counter register increments a soft counter (Figure 3.21(a)).




(a) Nettime format

(b) Cycle Counter reset

Figure 3.21: Cycle counter format

Each time a bus reset occurs, the software part of the "nettime" is reset and each IEEE-1394 interface resets its cycle counter (Figure 3.21(b)).

The application should manage the transition between these independent global times (before and after reconfiguration) using the local time of the node it is running on. By this way, we can obtain a continuous global time using a fusion of the IEEE-1394 cycle counter (that is reinitialized each 128 seconds) and the local clock (continuous).

In the next chapter, we present the use of this accurate synchronisation for the design and implementation of distributed scheduling strategies.

### 3.6.2 Services localization

When a client (receiver) requests a service, SCOOT-R middleware ensures the localization of the active server (emitter) providing this service.

Every node maintains a table of services containing their description, including the node address, the port associated to the server, the size of data, names of data types and parameters of the temporal contract. All the services are identified by their names. This table is updated by the servers and emitters registration algorithm and is reinitialized at each global or partial reconfiguration.

This approach is effective for networks with relatively modest size. Our implementation uses the IEEE-1394 bus, the number of nodes is limited to 63, which is sufficient for a wide range of applications.

### 3.6.3 Registration algorithm

A server or emitter tries to register when its quality is greater than that of the current registered server or emitter or when there is no registered server or emitter for the service.

If the current registered server or emitter decrements its quality factor, it sends the same registration message as if it really registers. This way, it informs all the network of its degraded quality. If it is no more the server or emitter having the best quality coefficient for this service, the better one will replace it (Figure 3.22).

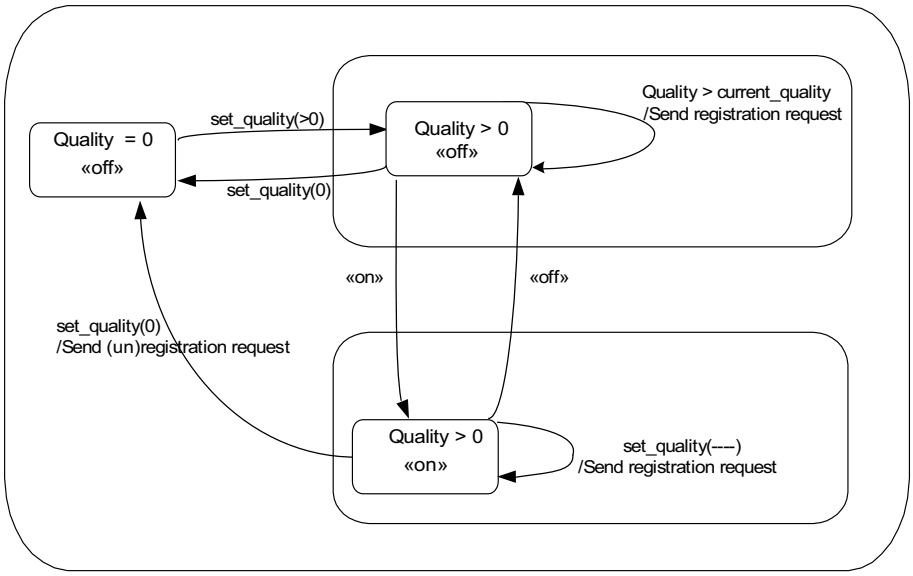



Figure 3.22: Statechart of the registration algorithm

We associate to each registration message a network date (timestamp) and we initiate the diffusion of the registration message to all the nodes by the diffusion mechanism described in section 3.6.4. To each reception of a registration message we apply registration algorithm outlined above (Algorithm 3.1).

The algorithm 3.1 selects the most recent registration message having the maximum "timestamp" date. If the message has a "timestamp" date equal to the current one (this may occur only when the requests are coming from different nodes), the algorithm selects in this case the message that have the highest site number (note that the nodes are numbered from 0 to MAX_NODES).

**Algorithm 3.1** Registration algorithm

```
INPUT : We examine the registration message of a service :
if "timestamp" of the message ≥ current "timestamp" then
   Set the new message as the active registration message; return;
else if "timestamp" of the message = current "timestamp" then
   if node number of the message ≥ local node then
      Set the new request as the active registration message ; return;
   end if
end if
```

This mechanism permits:

- that only one server is qualified by service, or rather, if there are several servers for the same service, one server will be activated in a bounded time (the server having the best quality will be activated) and all the others are stopped;
- that the server is recognized by all the clients being able to communicate with the server site in a bounded time (in absence of permanent communication error);
- that if a server module able to render the service exists, a server is qualified in a bounded time whatever was the former failures.

In order to maintain consistent state amongst the registered services, the service registration must be performed after waiting a certain amount of time since the receiving of the registration message. i.e., when a node receives a new registration message, it schedules to register the service in a future time. This delay permits to guarantee that all the other nodes have received this registration message and thus a coherent decision may be made. Unfortunately, this leads to a large delay of *service unavailability.*

The other solution is to register the service immediately after the reception of the registration message. By doing that, there may be a *transient* inconsistent states between the services. On the other hand, this leads to an immediate availability of the service.

Given the application's timing constraints, the designers may use one of these two techniques. In our implementation, we have chosen the second method to ensure a fast availability of the service and given that our applications are not critical from this point of view and do not require a deterministic behavior in case of failovers (service replacement).

In order to avoid the over use of the network and to bound the CPU consumption for service registration, each node waits for a sufficient delay between two registration messages. This solution is efficient on the IEEE-1394 bus, especially when all servers send their registration message after a bus reset.

### 3.6.4 A safe diffusion mechanism

SCOOT-R needs absolutely that any message describing a server registration (or unregistration) is received by all the SCOOT-R nodes. Then it is sufficient to use the registration algorithm described above to ensure that all the nodes take the same decision. So, SCOOT-R needs a safe diffusion mechanism.

For our experimental implementation using the IEEE-1394 bus, we use a *specific* but *unproved* mechanism. It addresses the minimization of the number of registration messages to be diffused after

each bus reset. After a bus reset, each node broadcasts its registration message(s) for each service presented in this node.

For that, our diffusion mechanism uses the IEEE-1394 broadcast mode to send the registration (unregistration) message. The message includes also the sequence index of the message (integer that goes up from zero to NB_MESSAGES). This sequence number is incremented and added to the message header by the sending node.

Then, each node receives all the registration messages and notes them per node status. It waits for a sufficient delay (TIME_TO_ACK) before the sending of the acknowledge message. This waiting time allows the reception of several registration messages by the node. Then, a collective acknowledge may be performed. By this way, we can decrease the number of messages to be diffused by each node.

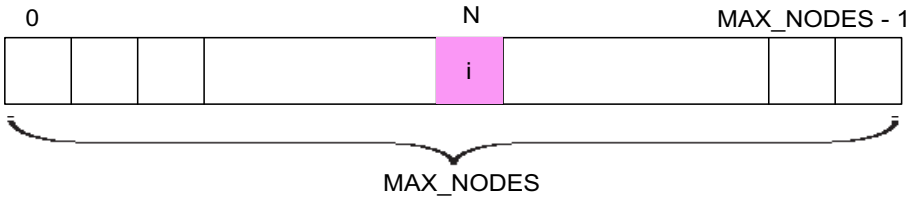



Figure 3.23: Broadcasted message format

The acknowledge message is broadcasted using the IEEE-1394 broadcast mechanism also. It contains a table of MAX_NODES entries that is the maximum number of nodes on the IEEE-1394 network (Figure 3.23). For example, i in the Figure 3.23 corresponds to the maximum sequence index of registration messages got correctly and in sequence from the node number N.

For example, if the node N broadcasted five messages (m0, m1, m2, m3, m4) and the receiving node has received only four messages as follows (m0, m1, m2, m4), so the receiving node has detected that there is a lost message (m3). In order to retrieve the lost message m3, the receiving node broadcasts a message that contains the maximum sequence index correctly received (as shown in Figure 3.23). For the scenario cited above, the broadcast message will contain 2. After reception of the acknowledge message, the node N resends the two messages 3 and 4 one by one using the guaranteed delivery point-to-point IEEE-1394 mechanism.

## 3.7 Defining application-level SCOOT-R objects

In SCOOT-R, the servers (emitters) and clients (receivers) are represented by objects and the communication between entities is performed via invocations on remote objects. There are five main classes in SCOOT-R to design an application as depicted in Figure 3.24.

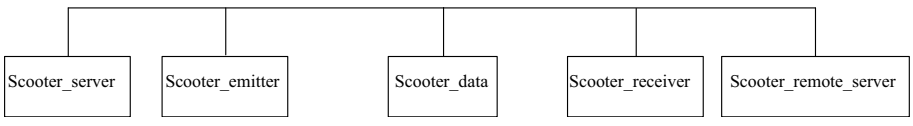



Figure 3.24: SCOOT-R main classes used in a user application

*Scooter_server* is the ancestor class which represents the server. All the servers used in a user application must inherit from this ancestor class. *Scooter_remote_server* is the ancestor class for the clients. A client implementation in the user application will inherit from this class. For the

emitter/receiver model, we have an homogeneous representation. Thus the *Scooter_emitter* class is the ancestor class for application emitters objects and *Scooter_receiver* for the application receivers objects. Finally, even the data exchanged between the clients and their servers are encapsulated in SCOOT-R objects. *Scooter_data* represents the ancestor class of any exchanged data between client and servers. The exchanged messages between the emitters and their receivers are represented by data buffers.

Thus, in order to create a new application, it is sufficient to inherit classes from one of the five ancestor classes and then to specify and redefine their functional and temporal specifications. i.e., by implementing the pure virtual methods of the main class.

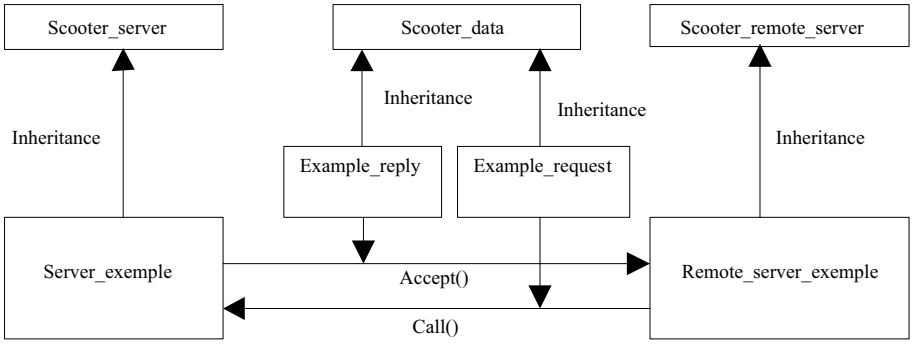



Figure 3.25: An example of a user application implementation by inheritance of SCOOT-R main classes

Figure 3.25 illustrates an example of a client/server application. In SCOOT-R, clients (receivers) and servers (emitters) exchange *named type* messages. SCOOT-R checks that clients (receivers) and servers (emitters) are type compatible. In an object-oriented approach, compatible is not identical: a server can accept a request sent by any client if this request inherits from the type of the request the server waits for. Symmetrically, a server can return to a client a reply message that inherits from the type of a message the client waits for. In SCOOT-R, the inheritance level for exchanged data is limited to two and it is defined at the compilation.

Thus, adding a new service SCOOT-R is done by defining the service, then describing the data structure exchanged between client (receiver) and server (emitter). Then, implementing at least one server (emitter) and writing the stub (on the client (receiver) side) to allow a client (receiver) to access the server (emitter).

### 3.7.1 Defining a server object

Creating an object from a class derived from the *Scooter_server* base class implements a server. The derived class *Server_example* has to implement the pure virtual methods from the base class.

For the source code shown in Listing 3.1, the methods "worst_case", "min_period" and "max_at_once_clients" describe the real-time contract this server offers to its clients.

The "worst_case" must be large enough so that the transaction always complete in this delay including the communication media access delay. If not, the server can be unregistered (it is considered as defective).

The "min_period" determines the maximum global requests rates and so the maximum CPU usage of this server. Thus, the CPU usage of this server can be bounded and a RMA analysis can be done for lower priority processes.

The "max_recover" specifies the maximum delay for the service to be re-established after a failure. This value is checked only for compatibility between clients and servers, but no check is performed on its respect.

The "max_period" allows the server to control the delay to forget a client. If there is any request from the client after "max_period", the server forgets the client and considers it as inactive. "max_-period" must be long enough so that client connection will be stable, but short enough for human management operations (a few seconds).

The "get_buffer_in()" and "get_buffer_out()" methods return prototypes of the "scooter_data" class the server waits for and returns back.

The "accept" method is the main server method which is called each time a client asks for the service, sending a request and waiting for a returned reply. The parameters are buffers provided by the SCOOT-R low-level software. The "ask" parameter points to a copy of the request sent by the client, and a copy of the "reply" addressed buffer will be delivered back to the client.

Listing 3.1: Server code example

```
#include "scooter_server.hh"
... ... ...

class Server_example : public Scooter_server
{
  example_request In;  example_reply Out;
  char * S; // local data for the server_example
public:  int nb_accept_call;
public:
  virtual RTIME worst_case() { return       100000 ; } // 0.1 milli
  virtual RTIME max_period() { return  ((RTIME)1000000000) *10 ; } // 10 sec
  virtual RTIME min_period() { return       100000 ; } // 0.1 milli - 10 kHertz
  virtual RTIME max_recover(){ return    100000000 ; } // 100 milli
  virtual scooter_data* get_buffer_in() { return &In ; }
  virtual scooter_data* get_buffer_out() { return &Out ; }
  virtual int max_at_once_clients(void) { return NB_CLIENTS; }

  Server_example( char* name ,char *s, pL_e_s L)  : Scooter_server( name , L )
  {
    init_server(); // now the server can be really started
  }
  virtual void accept( scooter_data *ask, scooter_data* reply)
  {
    /* Here is the source code of the server treatment */
  }
};
```

In order to start the server, the application must set its quality to a positive number. To stop the server, it is sufficient to set its quality to zero (Listing 3.2).

Listing 3.2: Initializing and Launching of the server

```
my_scootr_handle = find_l_e_s("OHCI-1394", 0 ); // the first OHCI-1394 adapter
if( my_scootr_handle == NULL ) // nothing to do, network unavailable.
  {
    printf(  "Real-time LES - network unavailable\n" );
    return -1;
  }

A = new server_example( my_scootr_handle );
A->set_quality( 2 ); // Now, the server is operational

/* To stop the server */
A->set_quality( 0 );
```

### 3.7.2 Defining a client object

The *Remote_server_example* (client stub) inherits from the client ancestor class *Scooter_remote_server*. The pure virtual methods in the base class have to be provided in the same way than for the server class. For the source code shown in Listing 3.3, "get_buffer_request()" and "get_buffer_reply()" methods return prototypes of the "scooter_data" class the client sends and returns back.

The "worst_case" must be large enough so that the transaction always complete in this delay including the communication delay. If not, the server is unregistered. SCOOT-R checks if this value is greater than those in the server declaration to start the client transaction.

Listing 3.3: Client code example

```
#include "scooter_remote_server.hh"
      ... ... ...

      class Remote_server_example : public Scooter_remote_server
      {
    example_request Request;
    example_reply   Reply;
            public:
    virtual RTIME worst_case() { return      1000000 ; } // 1 milli
    virtual RTIME min_period() { return       500000 ; } // 500 micro
    virtual RTIME max_recover(){ return    200000000 ; } // 100 mill

    virtual scooter_data* get_buffer_request() { return &Request ; }
    virtual scooter_data* get_buffer_reply() { return &Reply ; }

    Remote_server_example( char* name ,pL_e_s L) : Scooter_remote_server( name , L )
    {
      init_remote_server();
    }
      };
```

Let's recall that this object is not the client; it is the object the client uses to access the service (client stub). Hence, in order to communicate with the server object, the application developer has to setup a task which initiate this client/server communication. For the source code shown in Listing 3.4, the task invokes the "call" method of the *Remote_server_example* object with a request parameter ("res").

When the call completes, it gets back an exception value. If "Exception" value is not NULL, the transaction does not complete successfully and the value of this exception defines its type (e.g., timeout, server_down, etc.).

Listing 3.4: Client task code example

```
void client_task_code(int t)
  {
    Message *Exception; // object describing the type of the returned exception
    scooter_data *res; // here i get back the address for the returned object.
    /* creating the Scooter_remote_server object (client stub) */
    A = new Remote_server_example(my_scootr_handle);
    Exception = A->call_log( &res ); // initiates the client transaction
      if( Exception == NULL )
        { // fine
          printf("transaction succeed");
        }
      else
       {
        printf("There is an exception occurred");
       }
  }
```

### 3.7.3 Defining an emitter object

As the server side, creating an object from a class derived from the "Scooter_emitter" base class implements an emitter. The derived class has to implement the pure virtual methods from the base class.

For the source code shown in Listing 3.5, the methods "period" and "jitter" describe the real-time contract that the emitter offers to its receivers. The receiver declares that it will respect the average "period" by keeping also a local variation smaller than the "jitter".

Listing 3.5: Emitter code example

```
class Emitter_example : public Scooter_emitter
{
   example_isoch *Out; // data type of the sending data objects

public:
  virtual RTIME period() { return((RTIME)1000000); } // 1 milli sec
  virtual RTIME max_recover(){ return   100000000 ; } // 100 milli
  virtual RTIME max_jitter(){ return   400000 ; } //  400 micro
  virtual scooter_data* get_buffer_out() { return &Out ; }
  Emitter_example( char* name ,  pL_e_s L)  : Scooter_emitter( name , L) ;
  {
      init_emitter(); // now "this" is ready and the emitter can be really started
  }
};
```

In order to setup an emitter, the user application must implement a task which call the "deposit" method of the emitter with respect to the emitter *period* and *jitter* as shown in Listing 3.6.

Listing 3.6: Emitter task code example

```
    void emitter_task(int t)
    {
      Emitter_example *mthis = new Emitter_example(my_scootr_handle);
      mthis->set_quality( 2 ); // starts the emitter
      while( ! FINI ) {
        mthis->deposit( data *);
        Sleep(mthis->period());
      }
    };
```

### 3.7.4 Defining a receiver object

The pure virtual method in the base class have to be provided in the same way than for the emitter class. Each time a complete message is received from an emitter, the "consume" method is called (Listing 3.7).

Listing 3.7: Receiver code example

```
class Receiver_example : public Scooter_receiver
{
      example_isoch *In; // data type of the received data objects

public:
      virtual RTIME min_period() { return       500000 ; } // 500 micro
      virtual RTIME max_period() { return      1500000 ; } // 1500 micro
      virtual RTIME max_recover(){ return    100001000 ; } // more than 100 mill

      virtual scooter_data* get_buffer_in() { return In; }
      Receiver_example( char* name , pL_e_s L) : Scooter_receiver( name , L )
      {
        init_receiver();
```

```
        }
    virtual void consume( int channel)
    {
    // here, we process the incoming data objects for the specified channel
    }
    };
```

## 3.8 Performances

This section presents some indications on the performance of SCOOT-R middleware. The values presented here are obtained for real experiments based on a test bank that we have performed. During the test, the network is free from other client/server or emitter/receiver activities and free also of any non SCOOT-R activity. The performances are measured while running PCs 800 Mhz Pentium, using RTAI/Linux kernel and connected to the IEEE-1394a bus (400 Mbit/s). The following Table 3.3 summarizes the performance baselines in term of communication delays.

| | Kernel Mode (RTAI) | LXRT Mode (Linux real-time) |
|---|---|---|
| average delay (local) | 8 $\mu$s | $\leq$ 15 $\mu$s |
| worst delay (local) | $<$ 50 $\mu$s | $<$ 60 $\mu$s |
| average delay (remote) | $\leq$ 50 $\mu$s | $\leq$ 50 $\mu$s |
| worst delay (remote) | $<$ 100 $\mu$s | $<$ 100 $\mu$s |
| service registration delay | $\approx$ 120 $\mu$s | $\approx$ 120 $\mu$s |

Table 3.3: SCOOT-R performances (communication delays) – client/server model

The test bank is conducted using two different modes of SCOOT-R. The first mode is the *kernel* mode, in this case the clients (receivers) and servers (emitters) are implemented as kernel modules of the RTAI OS.

The other mode of implementation is the *LXRT* mode. Here, the clients and servers are implemented as standard Linux processes.

In order to obtain the delays depicted in Table 3.3 and to evaluate SCOOT-R, we conducted several interactions client/server ($\approx$ 10000 transactions). In *local* experimentation, the client and its server are located on the same computer. The computer does not running other SCOOT-R activities. In the *remote* experimentation, the client and its server are located on two different computers connected by the IEEE-1394 bus. The computers also do not run other SCOOT-R activities.

As shown in Table 3.3, the average client/server transaction delay for local communication (client and server located on the same computer) is less than 8 $\mu$s (this is the CPU overhead for the SCOOT-R stack - about 10000 cycles per transaction) when the client and the server are implemented as RTAI kernel modules.

The average client/server transaction delay for local communication is less than 15 $\mu$s (this is the CPU overhead for the SCOOT-R stack - about 15000 cycles per transaction) if the client and the server are implemented as RTAI-LXRT Linux processes.

The worst case client/server transaction delay for local communication is less than 50 $\mu$s and the average client/server transaction delay for remote communication is less than 50 $\mu$s (it looks like there is a relatively long hardware delay on the IEEE-1394 adapters).

The maximum frequency for the emitter/receiver paradigm is 8 kHz (limited by the IEEE-1394 basic cycle of 125 $\mu$s).

The recovery time when a server or an emitter is registered or removed or replaced by another one is very short too. 120 $\mu$s is the worst observed case without other communication activity or servers/emitters registering activity.

## 3.9 Typical automotive application involving SCOOT-R

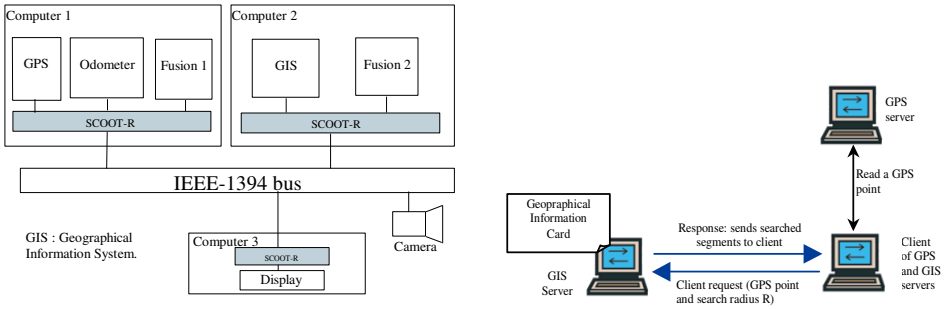
In this section, an overview of a real automotive application is given to illustrate the utility of our SCOOT-R services for distributed and real-time applications. The application (Figure 3.26(a)) presented here is a small subset of the RoadSense application. It permits the real-time accurate positioning on a digital map GIS (Geographical Information System) [NB02].

### 3.9.1 Presentation of the application

The application provides an accurate estimation of position for a vehicle relative to a digital road map. Many modern in-vehicle navigation and safety applications require real-time positioning of the vehicle with respect to a given set of digital map data. Real-time positioning allows the driving assistance component to accurately depict the position of the vehicle on the map, facilitates operations such as route calculation, supports ADAS applications such as Adaptive Cruise Control (ACC), adaptive lighting control, collision warning and lane departure warning. For driving assistance applications, the positioning component is of crucial importance to reach the ADAS attributes stored in the database, like the radius of curvature, the width of the road or the speed limits.

### 3.9.2 Internal structure of the application's components

For our application and in order to compute the vehicle position with the best accuracy and reliability, GPS and Odometer sensors are necessary for the vehicle localization. These sensors and other tasks like the fusion and GIS (Geographical Information System) are encapsulated in SCOOT-R components (Figure 3.26(a)).




(a) General architecture of the application (b) Communications client/server, GPS - GIS

Figure 3.26: communication architecture of the application

The "GPS" sensor component permits to acquire GPS data frames through the serial bus and to

provide them in real-time on the network as a SCOOT-R server. We use a differential GPS with a precision about 4-5 meters at 1 Hz. It provides the GPS sensor data as a SCOOT-R server.

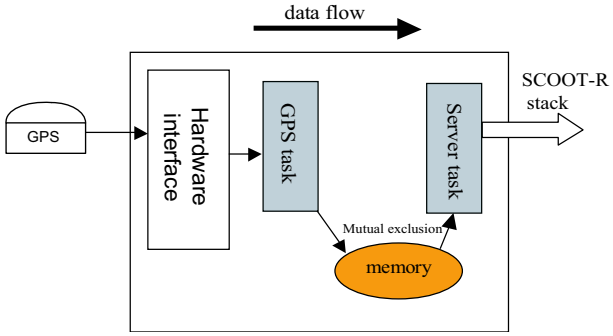



Figure 3.27: GPS sensor component structure

As shown in Figure 3.27, the GPS component acquires GPS data from the hardware interface of the sensor. It includes two tasks, the "gps task" that performs the run-time data acquisition and that stores them on a local memory variable protected by a mutual execution. The "server task" will recuperate the last GPS data stored in the local memory each time a request arrives from a client and it will return the GPS data as a SCOOT-R server's reply using the SCOOT-R stack.

The "odometer" sensor component acquires through a PC-card the speed and the distance flowed by each wheel. It provides the odometer sensor data as a SCOOT-R server.

The "GIS" component searches in a circle of radius R and center C (measured by the GPS) all linear road segments and provides them in SCOOT-R data-compliant format (Figure 3.26(b)). It reads the GIS information from the local database and then provides them as a SCOOT-R server. Let's note that the GIS component has a high worst case CPU overhead (1.5 seconds).

The first fusion component (fusion1) combines GPS and Odometer data for accurate positioning that is kept between the GPS 1 Hz messages and during the GPS masking that may remain for several seconds. In order to improve the precision, the vehicle position is mapped to the GIS map by the second fusion component (fusion2).

The "fusion2" component asks for GIS information in asynchronous way and uses for its position computation a cached part of the data map without waiting for the last updated data from the "GIS" server. The set of road segments returned by a request to the "GIS" server covers a sufficient radius that may be used by many several consecutive map-matching operations.

The components "fusion1" and "fusion2" perform asynchronous calls to the different servers that need to achieve their computations. These computations are performed at a fixed frequency (this is the frequency of the fusion algorithm). Moreover, the "fusion1" and "fusion2" components perform a call to the "display" server each time a new position is computed.

Implementing a SCOOT-R client/server service is done by defining new C++ classes for exchanged data, server and client sides stubs. These classes inherit from base classes of the SCOOT-R library (§3.7). Below, we detail the specifications of the SCOOT-R real-time contract and its use in the case of the "GPS".

To create the "GPS" server, a descendant class "server_gps" is created from the generic class "scooter_server". The method get_buffer_in() defines the object type received from the client and get_buffer_out() defines the object type returned by the server. The "accept" method is called at every transaction initiated by any client of the "GPS" service. In this example, it consists in providing

to the client, the information on the actual position and speed of the vehicle.

The server definition also includes needed functions to supply the real-time specifications listed in Table 3.1. For the "gps_server" implementation, "Max_clients" will be set to 10, "Min_Period" to 100 ms and "Max_Recover" to 200 ms. "Min_Period" permits to compute the CPU and network load taken by this client.

At the application level, an instance of the previously defined server class is created, then one has to manage properly the quality of the server using the set_quality() method.

In order to use the server defined previously, a new class is defined by inheritance from the basic client class. For the "gps_client" side, "Min_Period" to 100 ms and "Max_Recover" to 200 ms.

The "worst_case" of the "server_gps" and its client may be set after a worst case analysis of the network (described later in this section), or it may be assigned before this worst case analysis but it must be sufficient small in order to ensure a correct operation of the application.

Once a local object is created as an instance of the previous "remote_server_gps" class, the "fusion" or "GIS" applications use the call() method from this object to get back the GPS data from the "GPS" server. In the case of GPS masking, the "GPS" server turned off and the "SERVER_DOWN" exception will occur.

Let'us recall that "fusion1" and "fusion2" components compute redundant data and they implement SCOOT-R replicated servers. When the "fusion1" server does not respond to the client requests, it will be considered as defective and the client transaction will fail. The server will be replaced by the "fusion2" server. The client should wait until this replacement process has been completed before initiating another transaction.

The commutation from "fusion1" to "fusion2" server is done in a bounded time without disturbing the clients invocations on "fusion1" server. So, when the "fusion2" server is ready to operate, the client may access it and continue its transactions.

### 3.9.3 Timing constraints of the application

We need that the global imprecision degradation to be less than one meter (GPS imprecision + 1 meter). Let's consider a vehicle with a nominal speed of 30 m/s; note that one-meter corresponds to 33 milliseconds at this speed. This precision should be preserved between the GPS messages and during GPS masking. Using the SCOOT-R time-stamping mechanism for a distributed environment, the degradation of the time-stamping does not exceed 10 $\mu$s that is insignificant regarding the global imprecision (33 milliseconds). Thus, the timing constraints of the application require a "display" frequency less than *25 milliseconds* and a global delay from the delivered data to the display component less than *33 milliseconds*.

The access by the "fusion1" component to the odometer data and for the "fusion2" component to the data produced by the "fusion1" component should be done in a strictly bounded delay for the application to operate properly. More critical even is the access to the "display" server, as unexpected delay can't be taken into account by a predictive algorithm. If each of these data communication is less than 1/100 second (the transaction delay to be verified), our application delivers correct results.

In the zoomed zone of the Figure 3.28, a uniform speed is represented by the oblique dashed line (45 degree). GPS data are acquired at 1 Hz while the odometer data are acquired at 100 Hz. We present at the Y-axis the imprecision intervals. Between the GPS acquisition time (tgps) and the odometer acquisition time (tod), we have the real value of the vehicle position. After tod, we do not have a valid measure of the position, and the vehicle model changes dynamically (region between upward and

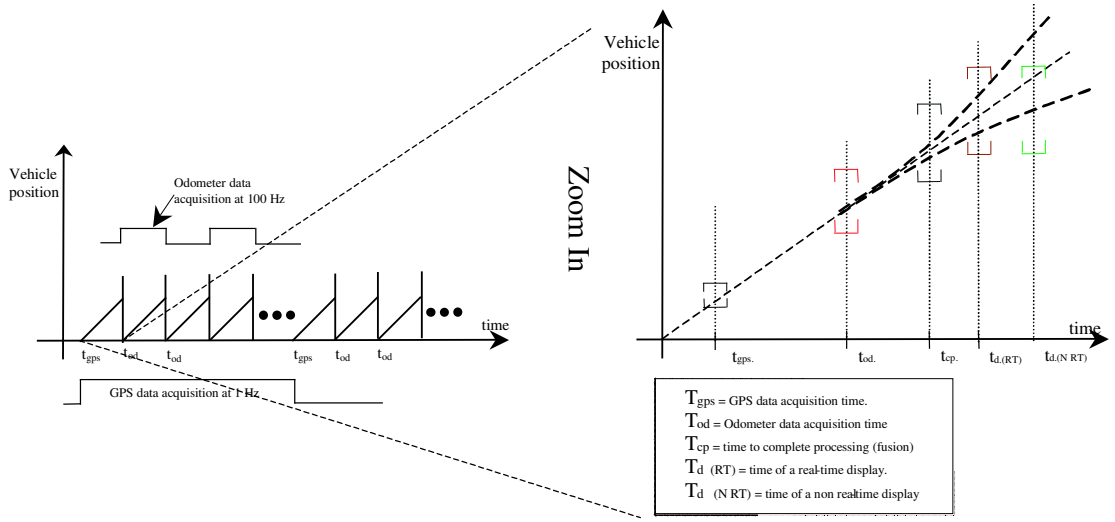



Figure 3.28: Position imprecision and computation delays

downward curved dashed lines).

The dynamic model of the vehicle uses odometer data to compute its position at instant tod. We add the imprecision of the GPS data acquisition time to the imprecision of the GPS module ($\approx$ 5 meters).

The imprecision of odometer data and of data acquisition time degrades the results quality. These computations are achieved and displayed slightly later; so predictive model is used to provide results at the display time rather than at the acquisition time.

### 3.9.4 Worst case time analysis for the distributed application

In nominal operation and for each computer, a local RMA (Rate Monotonic Analysis) is performed to determine the worst-case behavior for each activity. We take into consideration the maximum number of clients for each server. So, each computer is validated as a hardware component. The following tables present the results of the RMA analysis for each computer. Let's recall that this analysis consider the maximum charge of each computer, i.e., the maximum clients number supported by each server and not the number of clients deployed effectively.

| | WORST DELAYED | CPU TIME PER ACTIVITY | WORST DELAY TO COMPLETE |
|---|---|---|---|
| ISR (Interrupt Service Routine) for timer, IEEE-1394, serial line (GPS) and high priority task (Acquisition odometer) | 0 | $\leq$ 500 $\mu$s at 100 Hz | 500 $\mu$s |
| Odometer server | 500 $\mu$s | 80 $\mu$s (per transaction) * (8 clients) = 400 $\mu$s | 0.9 ms |
| Fusion 1 computation and server | 0.9 ms | 1 ms (fusion1 computation cost) + 50 $\mu$s * (8 clients) = 1.4 ms | 2.3 ms |
| SCOOT-R overhead | 2.3 ms | 200 $\mu$s | 2.5 ms |
| GPS server | 2.5 ms | 60 $\mu$s * (8 clients) $<$ 500 $\mu$s | 3.1 ms |

Table 3.4: RMA for computer 1

| | WORST DELAYED | CPU TIME PER ACTIVITY | WORST DELAY TO COMPLETE |
|---|---|---|---|
| ISR for disk activities | 0 | $\leq$ 10 $\mu$s at 2 kHz | 10 $\mu$s |
| Fusion 2 server | 30 $\mu$s | 2 ms (fusion2 computing cost) + 50 $\mu$s (per transaction) * (8 clients) = 2.4 ms | $\leq$ 2,5 ms |
| GIS server | 2.5 ms | $\leq$ 1.5 seconds (one client) | $\leq$ 1.6 seconds |

Table 3.5: RMA for computer 2

| | WORST DELAYED | CPU TIME PER ACTIVITY | WORST DELAY TO COMPLETE |
|---|---|---|---|
| ISR for VGA card | 0 | $\leq$ 10 $\mu$s at 4 kHz | 10 $\mu$s |
| Display server | 10 $\mu$s | $\leq$ 2*(50 $\mu$s + 5 ms) | 10.11 ms |

Table 3.6: RMA for computer 3

Knowing the list of clients and servers and the size of messages sent on the network, we can compute the worst time for each node to send a message.

Thus and for each computer, we compute the maximum number of asynchronous messages to be emitted by this computer. This number (maximum) can be computed by the following equation:

$$MAX\_ASY\_MSG\_TO\_EMIT = nb\_clients\ (for\ remote\ servers)\ + \sum(remote\ clients\ per\ server) + 1 \qquad (3.1)$$

As shown in Equation 3.1, we consider only the number of *remote* clients and also only the servers of *remote* clients. It is obvious that the communication between *local* clients and servers does not need a media access.

Using the Table 3.7 we can compute easily the maximum number of messages to be emitted for each computer. The one appeared in the equation 3.1 corresponds to the registration message emitted by the node. For example, for the computer 1, we have: MAX_ASY_MSG_TO_EMIT = 2 (two clients for remote servers "GIS" and "display") + 1 (one server for remote client "fusion1") + 1 (for registration message) = 4.

| COMPUTER NAME | CLIENTS | SERVERS | MAX_ASY_MSG_-TO_EMIT |
|---|---|---|---|
| Computer 1 | GPS, Odometer, GIS, Display | GPS, Odometer, fusion1 | 4 |
| Computer 2 | Display, fusion1 | GIS, fusion2 | 4 |
| Computer 3 (display) | No clients | Display | 3 |
| Camera | Isochronous data flow (constant bandwidth usage) | | |

Table 3.7: Clients and servers per computer

The following equation permits to evaluate the media access time:

$$MAT \leq nb\_nodes * MET \qquad (3.2)$$

Where MAT = Media Access Time, and MET = Maximum Emission Time. Let's note that there is no priority on the bus access but a fairness interval mechanism allows each node to emit in a bounded time. In our example, we have four computers in the network. So, from the previous:

$$MAT \leq 4 * MET \tag{3.3}$$

Where:
MET = 10$\mu$s + Max_Length/speed;
Max_Length = Maximum size of messages in bytes;
Speed = Transmission speed on the IEEE-1394 bus in Mb/s;

In case of messages with maximum size of 1000 bytes and a bus speed of 400 Mb/s, we obtain: MET = 12.5 $\mu$s and MAT = 50 $\mu$s.

Let's note that the 10 $\mu$s in the MET formula corresponds to the arbitration period on the IEEE-1394 bus.

| COMPUTER NAME | CLIENTS | SERVER RESPONSE TIME | MAX_ASY_MSG_-TO_EMIT (local node) | MAX_ASY_MSG_-TO_EMIT (remote node) | TRANSACTION TIME |
|---|---|---|---|---|---|
| Computer 2 | GPS | 3.1 ms | 0 | 0 | 3.1 ms |
| | Odometer | 0.9 ms | 0 | 0 | 0.9 ms |
| | GIS | 1.6 seconds | 4 | 4 | $\leq$ 1.7 seconds |
| | Display | 10.11 ms | 4 | 3 | $\leq$ 11 ms |
| Computer 2 | fusion1 | 2.3 ms | 4 | 4 | 2.7 ms |
| | Display | 10.11 ms | 4 | 3 | $\leq$ 11 ms |
| Computer 3 | No clients | | | | |

Table 3.8: Computation and network delays

In order to compute the transaction time in Table 3.8, it is sufficient to compute the maximum number of messages to be transmitted at the local and remote nodes and to add the server response time (worst case) computed by the local RMA analysis on each computer. For example to compute the transaction time of the "display" client on computer 2, we have to add the MAX_ASY_MSG_-TO_EMIT on computer 2 (local node) to the MAX_ASY_MSG_TO_EMIT on computer 3 (remote node) and to add the worst response time of the server "display" (10.11 ms), so we obtain, transaction time = (4 + 3) * 50 $\mu$s + 10.11 ms $\leq$ 11 ms.

Moreover, the arbitration mechanism on the IEEE-1394 bus is guaranteed by a fairness interval for asynchronous transactions. The fairness interval ensures that each node wishing to initiate a transaction gets fair access to the bus.

As shown in Table 3.8, the two "display" clients transactions are less than *25 milliseconds*. In addition, the time from the data source to the "display server" (while taking into account the time to acquire the data) are less than *33 milliseconds*. For example and as shown in Table 3.8, the transaction time of the "odometer", "GPS", and "fusion1" components are less than 8 milliseconds (33 milliseconds - 25 milliseconds for the display).

Moreover, let's remember that the *1.6 seconds* in the "GIS" server's response time does not put any difficulty to the application requirements as the "GIS" server returns a set of segments widely sufficient to perform several consecutive map-matching operations. So, the application's temporal requirements are fulfilled.

Now the worst case analysis is made for the components of the network, we can set the "worst_case" values of the different servers to achieve the dynamic diagnostic feature of SCOOT-R.

## 3.10 Conclusion

This chapter has given a presentation of the architecture and current status of the SCOOT-R project developed at out laboratory, aiming at building distributed real-time applications. The solution rests on a set of basic services built as a middleware layer above a real-time kernel. Essential protocols for synchronisation of clocks, replication and reliable diffusion are implemented. Special attention was given to the communication subsystem since it is a common resource to the middleware services developed. The communication subsystem is based on client/server and emitter/receiver models.

The contribution of the thesis has related to the evolution of the temporal SCOOT-R model (temporal contract and rules). Then, we have contributed to the development of the *emitter/receiver model* that is added to the basis client/server model to provide a complete distributed solution with the support of data flow applications (e.g., real-time image acquisition and processing, analog signal acquisition, etc.). Finally, the thesis has contributed to the *development methodology* of user applications and the *worst case analysis* of the distributed system.

The redundancy management in SCOOT-R system does not require specific efforts of design and development to dynamically replace a server or an emitter or to activate a redundant function. This enables an evolution of the services without interruption.

Given our participation to the RoadSense project whose main goal is to elaborate driver behavioral indicators (or metrics) for safety, comfort and support assessment, and given the large amount of data to be proceeded and the distribution of calculation on many computers have lead us to the use of SCOOT-R as the software architecture of our automotive applications.

The initial development of SCOOT-R does not consider the dynamic scheduling of tasks and messages. The messages are propagated using the FIFO scheme and the tasks are scheduled by the fixed-priority RTAI scheduler. In the next chapter, we present our contribution to enhance SCOOT-R by incorporating integrated tasks/messages scheduling.

# *Real-Time Scheduling*

## Abstract

Many research works have been conducted to schedule a set of tasks in a system with a limited amount of resources such that all tasks will meet their deadlines. However, research on real-time scheduling has experienced a major shift during the last years, from *local* to *distributed* scheduling.

Moreover, *middleware* in a distributed real-time system can provide the infrastructure and mechanisms required to perform the necessary scheduling (local and distributed) strategies. Thus, it provides the required end-to-end support for various real-time quality of service (QoS) aspects, such as bandwidth, latency, jitter, and dependability.

We present in this chapter distributed scheduling strategies that permit to ensure end-to-end real-time QoS capabilities in a middleware environment. Using our SCOOT-R client/server middleware presented in chapter 3, we compare static priority scheduling, with and without priority inheritance with the distributed EDF scheduling. We show also the easiness to take into account the criticalness constraints with EDF scheduling strategy in a distributed environment.

## Contents

## 4.1 Introduction

Scheduling algorithm is an important key to be considered in the design of a real-time system. A *scheduling algorithm* defines the set of rules that determine the execution order of tasks at any instant.

To a scheduling algorithm, one associates generally a *schedulability test* that permits to verify if all the real-time tasks will always meet their deadlines. Thus, the objective of an optimized scheduling is to define how the different actions of the tasks have to be performed with the objective of meeting the temporal constraints with the minimum of resources.

In the case that all information regarding the state of the system as well as the resource needs of a task are known (for example centralized systems), an optimal assignment can be made based on some criterion function [CSR89][dOdSF00][Tin93]. Examples of optimization measures are minimizing total task completion time, maximizing utilization of resources in the system, or maximizing system throughput. In the event that these problems are computationally infeasible (the case of distributed system), suboptimal solutions may be tried [LRWK04][MMMM01][MMM00]. The goal of a suboptimal solution is to find a feasible schedule for the distributed system with a sufficient robustness regarding the system configuration change or system failure.

A scheduling algorithm is often based on a *task model* that defines the set of constraints and rules, that must conform to the tasks. It depends also on an *architectural model* that describe the hardware architecture of the underlying system (e.g., distributed system, multiprocessors, etc.).

Middleware in a distributed real-time system can provide the framework and mechanisms required to perform the necessary scheduling. In such a distributed system, each endsystem may provide its own local scheduling mechanisms. However, the middleware must take into account the entire system, and provide coordinated scheduling information to the individual endsystems, which will enforce the globally determined scheduling decisions.

The global scheduling decisions that are made in a distributed real-time system include where to allocate service requests, how to provide scheduling information to local endsystems, and how to handle overload that can cause QoS failures in the system. The middleware, by defining a common communication semantic on all the nodes, facilitates the definition and implementation of end-to-end scheduling policies. Typically and during a client/server transaction, the priority and/or deadline as well as the criticality of the server's processing task are related and inherited from those of the client task that initiates this transaction.

In this chapter, we begin by a brief presentation of the tasks models used by the most scheduling algorithms. Then we give an overview of static and dynamic scheduling algorithms by presenting briefly some typical examples in each category. Then we discuss some strategies of *distributed* real-time scheduling, and the structure of distributed schedulers (§4.6). Finally, we present our contribution on the development of distributed real-time scheduling strategies. These scheduling techniques are implemented using our middleware SCOOT-R (§4.7).

## 4.2 Tasks definitions

Throughout this section, we give some definitions that are commonly used to define a task in the scheduling literature.

Tasks form the logical units of computation for a processor. A single application program will typically consist of a set of tasks. Each task has a single thread of control. For multiple tasks on a single processor, the execution of the tasks is interleaved. With a system containing multiple processors

(i.e., multi-processor or distributed), the tasks may be interleaved over all the processors in the system.

**Tasks structure and arrival patterns**: Two distinct forms of tasks structure may be immediately defined: *periodic* and *aperiodic* tasks. *Periodic* tasks execute on a regular basis; they are characterised by their *period* and their required *execution time* (per period). The execution time may be given in terms of worst case execution time.

The activation of an *aperiodic* task is, essentially, a random event and is usually triggered by an action external to the system. Aperiodic tasks also have timing constraints associated with them; i.e., having started execution they must complete within a predefined time delay. Often these tasks deal with critical events in the system's environment and hence their deadlines are particularly important.

Most real-time systems have a mixture of *periodic* and *aperiodic* tasks. Mok [Mok83] assumes also a minimum separation time between two consecutive arrivals of aperiodic tasks and calls them *sporadic tasks*. Our applications presented later in this chapter use mainly this kind of tasks.

**Inter-task synchronisation**: The tasks can interact between them according to a pre-determined order or by an explicit *synchronisation* mechanism. Therefore, a precedence relation between tasks may result. This relation is *static*, because it is known *a priori* and does not evolve at run-time. It is usually represented by a graph of dependence.

The tasks can share other resources that the processor. A resource may be qualified as *shared* or *exclusive* resource. In a *shared* resource, several tasks may access it instantaneously. *Exclusive* resources force the tasks to use them in mutual exclusion.

**Worst case execution time**: A real-time scheduler requires the Worst Case Execution Time (WCET) of tasks. This information is needed for the scheduler's admission tests and subsequently limits a task's execution time duration. Modern processors include several features that make it difficult to compute the WCET at compile time, such as out-of-order execution, multiple levels of caches, pipelining, and data-dependent graph execution, etc. One way of approximating the WCETs is to perform testing of the system of tasks and use the largest value of computation time obtained during these tests. The problem is that the largest value seen during testing may not be the largest observed in the operational system. Another typical approach to determine a task computation time is by analyzing the source code [CP00][CP01b].

**Deadline and Laxity**: All real-time tasks define a *deadline* D by which they have to complete execution. Task's laxity may be defined as the *time-to-deadline* minus the remaining computation time. If R(t) denotes the remaining CPU computation time of a task T at time t, the laxity of the task T at time t is therefore defined as:

$$L(t) = D(t) - t - R(t), \forall t \geq 0. \tag{4.1}$$

## 4.3 Scheduling algorithms characteristics

Several scheduling algorithms have been developed and having different characteristics. Precise terminology is necessary to describe and evaluate *static* and *dynamic* scheduling strategies. Ghosh [Gho94] defines a taxonomy of scheduling algorithms that we have used and simplified in Figure 4.1.

There are two main categories of real-time scheduling algorithms: static [CSR89][dOdSF00][SK96] and dynamic scheduling [LRWK04][CSR89][RSZ89b][CP99][ABRW91].

*Static scheduling* algorithm has complete knowledge of the tasks model and its properties: deadlines, periods, worst case execution time, etc. It calculates schedules for the system by generating a table that includes the tasks execution order. Static scheduling algorithms require little run-time overhead

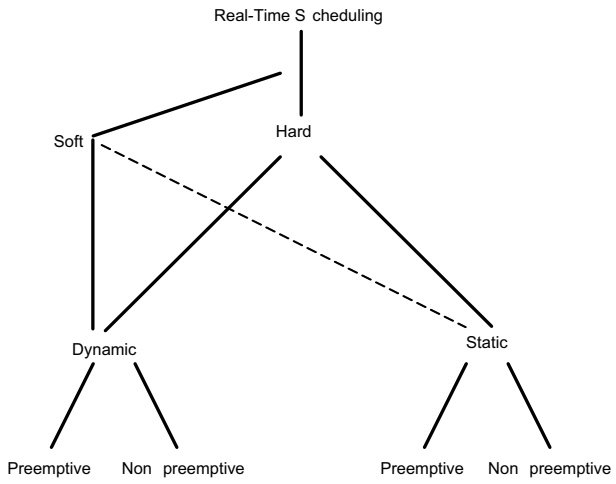



Figure 4.1: A simple taxonomy of some scheduling algorithms

and usually used for *periodic* or *sporadic* tasks with *fixed priorities*. Rate Monotonic (RM) algorithm [LL73] represents the major *static* scheduling algorithm.

*Dynamic scheduling* algorithm does not have the complete knowledge of the tasks model and its timing constraints. It calculates the execution order of tasks at run-time according to their arrivals and activation instants. Earliest Deadline First (EDF) [Gho94] is an optimal dynamic scheduling algorithm in resource sufficient environments where the system resources are available, to *a priori* guarantee that, even though tasks arrive dynamically, at any given time all them are schedulable. A dynamic scheduling algorithm may be used with all the kinds of tasks (periodic, sporadic, and aperiodic) with *fixed* or *dynamic* priorities.

A scheduler is called *offline* if all scheduling decisions are made prior to running the system. A table is generated that contains all the scheduling decisions for use during run-time. This relies completely upon *a priori* knowledge of tasks behavior.

*Online* scheduler makes scheduling decisions during the run-time of the system. The scheduling algorithm can be either *static* or *dynamic*. The decisions are based on both tasks model properties and the current state of the system.

A scheduler is called *preemptive* if it can arbitrarily suspend a task's execution and restart it later without affecting its behavior, except by increasing its elapse time. Preemption typically occurs when a higher priority task becomes runnable. Unfortunately in presence of inter-tasks synchronisation (e.g., resources sharing), this scheme may lead to an inter-tasks priority inversion, when a lower priority task suspends a higher priority task. This problem may be resolved by several methods, such as priority inheritance/ceiling protocol.

A *non-preemptive* scheduler does not suspend tasks. When the scheduler selects a task to be executed, it runs until its end of execution time. Thus, in this scheme, a higher priority task cannot suspend a lower priority task even when it becomes runnable. This approach avoids the priority inversion problem.

*Hybrid* systems are also possible. A scheduler may, in essence, be preemptive but allows a task to continue executing for a short period after the instant it should be suspended.

Finally, a scheduler may be labeled *local* or *global*. *Local* scheduler allocates the local processor to the set of tasks present in its node while respecting their timing and resource requirements. *Global* scheduler tries to guarantee the tasks constraints by considering and exploiting the processing capacities

of all the processors composing the distributed system.

After this brief presentation of the essential terminologies used in the scheduling literature. Hereafter, we present some typical examples of static and dynamic scheduling algorithms.

## 4.4 Static scheduling examples: the case of RM algorithm

As a typical example of static scheduling algorithm we present in this section the RM (Rate Monotonic) scheduling algorithm. The RM algorithm is a *static scheduling* algorithm based on periodic tasks. It makes the following assumptions about the tasks set:

- the requests are periodic for all tasks for which hard deadlines exist;
- all tasks are independent on each other. There exists no precedence constraints or mutual exclusion constraints between any pair of tasks;
- the deadline interval of every task is equal to its period;
- the required maximum computation time of each task is known *a priori* and is constant;
- the time required for context switching can be ignored.

The RM provides an optimal priority assignment and schedulability tests for systems scheduled with fixed priorities. The importance of these results is so significant that it has been a reference point for the later development of the scheduling theory.

Liu and Layland [LL73] showed that the assignment of priorities according to periods, that is the shorter the period, the higher the priority (Ti<Tj) Pi>Pj), is optimal in the sense that if there exists an assignment of priorities for which the system is schedulable, then it is also schedulable under this assignment. This is called the rate monotonic priority assignment. A sufficient condition of tasks acceptability may be given as follows:

$$\forall n, \qquad \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \tag{4.2}$$

where:

C i = Computation time;
P i = Period of the task;
n = number of tasks;

The RM scheduling algorithm may be implemented by an *offline* or *online* scheduler (preemptive or not) with fixed priorities. In the real world, most of the tasks are considered as *sporadic* (minimum separation time between two activations), and the RM algorithm may be applied to such tasks with the pessimist assumption on the tasks periods.

Moreover, the RM algorithm can be adapted to deal with aperiodic tasks also. The simplest approach is to provide a periodic task whose function is to service one or more aperiodic tasks. This periodic server task can be allocated the maximum execution time while continuing to meet the deadlines of periodic tasks.

As aperiodic events can only be handled when the periodic server is scheduled, the approach is essentially polling. The difficulty with polling is that it is incompatible with the bursty nature of aperiodic tasks. When the server is ready there may be no tasks to handle. Alternatively the server's

capacity may be unable to deal with a concentrated set of arrivals. To overcome this difficulty a number of bandwidth preserving algorithms have been proposed [CB97].

Static scheduling algorithms provide resource access guarantees at the cost of lower resource utilization. Certainly in safety critical systems it is reasonable to argue that no event should be unpredicted and that schedulability should be guaranteed before execution. This encourages the use of static scheduling algorithm.

To overcome the limitations of static scheduling, we have investigated the use of dynamic strategies to schedule SCOOT-R operations for applications with real-time QoS requirements.

## 4.5 Dynamic scheduling examples: EDF, MLF, and MUF algorithms

Dynamic scheduling has the potential to offer relief from some of the restrictions imposed by strict static scheduling approaches. Potential benefits of dynamic scheduling include better tolerance for variations in activities, more flexible prioritization, and better CPU utilization in the presence of non-periodic activities. However, the cost of these benefits has to be higher run-time scheduling overhead and additional application development complexity [RSZ89b].

Hereafter, we review briefly two well known *purely* dynamic scheduling algorithms, EDF and MLF algorithms [AB90]. In addition, we present the *hybrid* static/dynamic MUF [SK91a] scheduling algorithm discussed in section 4.5.3.

### 4.5.1 Earliest Deadline First (EDF)

EDF is a *dynamic* scheduling algorithm [AB90] that assumes a *preemptive* priority-based *online* scheduler. The tasks may be *periodic* or *aperiodic*. It dispatches the tasks at run-time based on their deadlines. The task with the current closer deadline is assigned the highest priority in the system and therefore executes.

The schedulability constraint is given as:

$$\forall n, \qquad \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{4.3}$$

where:

C i = Computation time;
P i = Period of the task;
n = number of tasks;

Hence, near 100 % processor utilisation is possible. This is a sufficient and necessary condition for schedulability in case of D i = P i (deadline = peirod). In [AB90], it has been shown that for an arbitrary tasks set in which task timing constraint is relaxed to allow deadlines to be different from periods, the condition in equation (4.3) is necessary but not sufficient. Thus, for arbitrary tasks, a sufficient condition of schedulability is:

$$\forall n, \qquad \sum_{i=1}^{n} \frac{C_i}{R_i} \leq 1 \tag{4.4}$$

where:

R i = Remaining execution time;

and a necessary condition is:

$$\forall n, \qquad \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{4.5}$$

A key limitation of EDF is that a task with the earliest deadline is dispatched, whether or not there is sufficient time remaining to complete its execution prior to the deadline. Therefore, the fact that a task cannot meet its deadline will not be detected until after the deadline has passed. Moreover, that task will continue to consume CPU time that could otherwise be allocated to other tasks that could still meet their deadlines.

Moreover, in the case of overloaded situations, the EDF performance degrades rapidly and may lead to an *avalanche* of non respected deadlines.

### 4.5.2 Minimum Laxity First (MLF)

MLF refines the EDF scheduling algorithm by taking into account task execution time. The task which has the least laxity is assigned the highest priority in the system and is therefore executed. The laxity of a task is defined as the deadline minus the remaining computation time. An executing task can be preempted by another whose laxity has decreased to below that of the running task. MLF uses the same tasks model of the EDF algorithm while adding the WCET of tasks (used to compute the task laxity). With the MLF algorithm, the scheduling constraint is again given by:

$$\forall n, \qquad \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{4.6}$$

Using MLF algorithm, it is possible to detect that a task will not meet its deadline *prior* to the deadline itself. If this occurs, a scheduler can reevaluate the operation before the CPU allocation.

A problem arises with this scheme when two tasks have similar laxities. One task will run for a short while and then get preempted by other and vice versa. Thus, many context switches occur in the lifetime of the tasks. This can result in "thrashing", meaning that the processor is spending more time performing context switches than useful work.

**Evaluation of EDF and MLF algorithms:** The main advantage of EDF and MLF is that they overcome the utilization limitations of RM algorithm. In addition, EDF and MLF handles non-periodic tasks comparably. On the other hand, a disadvantage of purely dynamic scheduling, as the EDF and MLF, is that their scheduling strategies require higher overhead at run-time. In addition, these purely dynamic scheduling algorithms perform poorly in case of insufficient resources situations; as the system becomes overloaded, the risk of an avalanche of deadline missing may occur.

To address these drawbacks, several scheduling algorithms have been developed to operate in insufficient environments (Spring algorithm [But05], LBESA [LRWK04], MUF [SK91a], etc.). In an asynchronous system, a scheduler is charged to respect tasks requests deadlines, but also to reabsorb the overloads or the failures consequences. In this case, it must guarantee the temporal requirements of the mandatory tasks for the process. It has to respect the deadlines and to consider the criticalness of the tasks in the case of overload. Thus, in order to deal with overload situations, several variants of EDF scheduling algorithm have been developed, as for example the Maximum-Urgency First (MUF) algorithm described below.

### 4.5.3 Maximum Urgency First (MUF)

MUF combines the advantages of the RM, EDF, and MLF algorithms [SK91a]. MUF supports the deterministic rigor of the static RM scheduling algorithm and the flexibility of dynamic scheduling algorithms like EDF and MLF. MUF can assign both static and dynamic priority to its tasks (Figure 4.2). The hybrid priority assignment in MUF overcomes the drawbacks of the individual scheduling strategies by combining techniques from each, as described below.

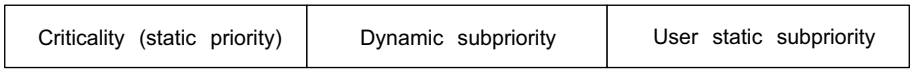



Figure 4.2: MUF priorities encoding

In MUF, tasks with higher *criticality* are assigned to higher *static priority* levels. This prevents tasks critical to the application from being preempted by non-critical tasks.

Ordering tasks by application-defined criticality reflects a subtle and fundamental shift in the notion of priority assignment. In particular, RM, EDF, and MLF show a rigid mapping from empirical task characteristics to a single priority value. Moreover, they offer little or no control over which tasks will miss their deadlines under overload conditions.

In contrast, MUF gives applications the ability to distinguish tasks arbitrarily. MUF allows control over which tasks will miss their deadlines. Therefore, it can protect a critical subset of the entire set of tasks.

A task's *dynamic subpriority* is evaluated whenever it must be compared to another task's dynamic subpriority and when the tasks have the same criticality. For example, a task's dynamic subpriority is evaluated whenever it is enqueued in or dequeued from a dynamically ordered dispatching queue. An example of such a simple dynamic subpriority function is the inverse of the task's laxity. Tasks with the smallest positive laxities have the highest dynamic subpriorities, followed by tasks with higher positive laxities.

In MUF, *static subpriority* is an application-specific and optional priority. It is used to order the dispatches of tasks that have the same criticality and the same dynamic subpriority. Thus, static subpriority has lower precedence than either criticality or dynamic subpriority.

## 4.6 Distributed Scheduling: a brief survey

In distributed real-time systems, end-to-end delay predictability for remote operations is essential for the critical parts of the system. Most distributed scheduling algorithms have two common features [MMMM01]: (1) a *media* scheduler (or *global* scheduler) between nodes and (2) a *local* scheduler for each individual node (Figure 4.3).

The local scheduling policy is often based on heuristics that efficiently determine which tasks to accept or reject.

In order to schedule real-time distributed systems, some works [MMM98] have adapted the scheduling algorithms with multiprocessor architecture to the distributed real-time scheduling after proper consideration of the communication scheduling. But given the inherent non-determinism of distributed and asynchronous systems, it is difficult to find an optimal scheduling algorithm satisfying all the timing and resource requirements of the applications. Thus, the use of multiprocessor scheduling techniques

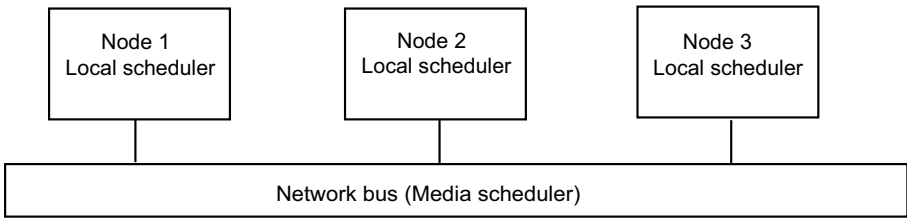



Figure 4.3: Local/Distributed scheduler

are not adapted to most of the distributed real-time systems. This leads to the use of integrated tasks (processor) and messages (media) scheduling techniques.

In distributed real-time systems, the worst-case response times of the communicating tasks are mutually dependent because of the messages exchanged by them. The distributed scheduling policies must take into account the synchronisation between the communicating tasks and also the scheduling policies of the messages on the network.

Tindell and Clark have proposed a Response Time Analysis method for hard real-time distributed systems called the *holistic analysis* [TC94]. This analysis permits to reproduce the schedulability analysis for fixed-priority tasks with arbitrary deadlines, in order to determine the worst-case response times of exchanged messages in a distributed environment. Thus, the holistic analysis permits to integrate the schedulability analysis for single processor systems with the timing analysis for hard real-time messages. Their approach is also useful for dealing with distributed systems based on several networks. Results are known for several kinds of networks like TDMA, 802.5, and CAN. The disadvantage of the holistic analysis is its pessimistic assumption while computing the worst-case delays.

In addition, one of the most important problems with *a priori* analysis for distributed fixed priority systems has been the complications introduced by communication costs: the delays for messages being sent between nodes must be accurately bounded, and the overheads due to communications must be strictly bounded.

The real-time *messages scheduling* aims to allocate the shared medium between several nodes so that the temporal constraints of the messages are respected. Thus, the messages scheduling constitutes a basic function of any distributed real-time system. There are two main classes of communication media, the *packet switching* networks and *multiple access* networks (§1.4). As we outline below, not all the messages generated in a distributed real-time application are *critical* from the temporal point of view. Thus, according to temporal constraints associated to the messages and to the network class used, two strategies of scheduling are employed:

***Best effort message transmission:*** *a message accepted for transmission is not necessarily transmitted by respecting its temporal constraints. This strategy is used to process the messages with soft or non real-time constraints.*

***Guaranteed message transmission:*** *each message accepted for transmission is transmitted by respecting its temporal constraints (except, obviously, in the event of failure of the communication system). This strategy is usually reserved for the messages with strict temporal constraints.*

In a distributed real-time system, both strategies may coexist, to be able to meet the needs for communication of various applications. With the emergence of distributed real-time systems, new needs for scheduling appeared: it is necessary, at the same time, to guarantee the respect of the temporal constraints of the tasks and those of the messages. As the messages scheduling takes into account similar constraints, real-time messages scheduling uses the same algorithms as the tasks scheduling. Thus, much of messages scheduling algorithms employ policies like Rate Monotonic [ABA+97][dOdSF00]

and Earliest Deadline First [RSZ89b][MMG04][RSZ89b].

Whereas the tasks may, in general, accept preemption, the transmission of a message does not admit preemption. If the transmission of a message starts, it is necessary to transmit all the bits of the message until the last, otherwise, the transmission fails. Thus, we must consider mainly non-preemptive scheduling algorithms or to use preemptive algorithms with the proviso of implementing a fragmentation of long messages in small packets and of reassembling them at the reception.

Several works have addressed the integrated tasks/messages scheduling. In [RSZ89b], the authors consider a distributed scheduling algorithms to deal with the timing and resources requirements. They use the notion of local and global scheduler to decide when and where a task must be executed. They made the focus on the tasks migration and the preallocation of resources. In [dOdSF00], the authors consider a set of periodic or sporadic tasks with fixed priority and precedence relations. They developed algorithms to transfer the precedence relations between distributed tasks into *release jitter* (initially introduced by Tindell for single processor scheduling) and then they apply the available schedulability test that is valid for independent tasks.

For distributed real-time systems based on the middleware execution support, we may relie on the communication semantic defined by the middleware to design and implement distributed scheduling strategies. Thus, a middleware can provide a good framework to design and evaluate distributed scheduling policies and thus ensures stringent real-time QoS to the distributed applications.

When using a real-time middleware to implement distributed scheduling algorithms, it should have policies and mechanisms in the underlying communication infrastructure that support resource guarantees. Since we are considering end-to-end timing constraints, the middleware must allow the user to express timing constraints, transfer them along the path of execution in the system, and always allocate resources in respect of these constraints.

In [SGHP97], the authors consider distributed scheduling algorithms based on transactional model. They propagate the scheduling policies and their parameters among the network using the CORBA client/server model. They obtain empirical results using the client priority propagation and the server declaration schemes.

### 4.6.1 Static versus dynamic distributed scheduling of communication resources

In a distributed environment, there are two main categories to schedule messages that correspond to two main major fieldbuses. The "static" and "dynamic" categories.

In static scheduling of messages, the communication and traffic is analyzed before run-time and configured in order to prevent run-time overloads. Hence, for the static configuration of a fieldbus traffic, only algorithms for non-preemptive scheduling can be used.

The dynamic scheduling of messages is based usually on transactional models. It consists of integrating messages/tasks using the timeliness propagation techniques.

Hereafter, we present and compare the main advantages and drawbacks of each category and the associated scheduling policies.

#### 4.6.1.1 Static scheduling of communication resources

Typical fieldbuses for static scheduling of messages are the TTCAN [CDKM00], Flexray and TTP/C [Kop00]. Almost all of these fieldbuses are based on Time-Triggered (TT) principle. TT architecture

is used for distributed real-time systems in safety-critical applications, such as computer controlled brakes, computer controlled suspension, or computer assisted steering in an automobile [H03].

In these distributed computing systems, a global time base is established and used for the generation of synchronized time triggers throughout the system. In a TT system, every task will periodically observe the state of its environment to determine whether a particular computational activity has to be performed. In a TT architecture, the scheduler has a complete knowledge about the system state before execution, e.g. release time, computation time, period, etc.

Static TT scheduling algorithms are usually qualified as *offline* and they are suited for safety critical applications and excel in temporal predictability. But the use of *online* algorithms for tasks has the advantage of robustness and flexibility. On the other hand, the TT buses usually lack of the bandwidth for high-level automotive functions (e.g., telematics, driving assistance functions, infotainment).

Using priority-based fieldbuses permits to built a global scheduler for the distributed system using the notion of *message priority*. The messages are scheduled *online* at run-time using the static priorities of messages. Thus, it is possible to implement an *online* media scheduler compliant with the RMA analysis for messages. A typical example of such network buses is the CAN bus.

#### 4.6.1.2 Dynamic scheduling of communication resources

While distributed static scheduling is well suited for safety critical applications, distributed dynamic scheduling approach is suitable for high-level applications that require more reconfiguration and that process large amount of data. Typical fieldbuses that may be used for dynamic scheduling are MOST [Gro00] and IEEE-1394 [And98] buses. They are based on a fair and concurrent media access and a collision avoidance mechanism that allow the access to media in a bounded time. These buses provide a distributed synchronisation technique, so a global time base may be obtained by a mixed hardware/software algorithm. The scheduling algorithms used here are "dynamic" and "decentralized".

A typical application of distributed scheduling techniques is the Advanced Driving Assistance System (ADAS), these applications are not mission-critical of the vehicle system and can be viewed as the second critical class after the safety critical one (e.g., engine and brake system control). However, they need a high-level of reconfiguration as they operate in dynamic and non-deterministic environments characterized by the unpredictable nature of the vehicle, road conditions and driving situations.

The scheduling in such distributed systems consists of the propagation and coordination of scheduling attributes for messages or rather transactions (client/server model) and then of pertinent use of this attributes by each local *online* scheduler. The scheduling algorithms reside at the higher application layer of the software architecture; it is implemented as a service of the software framework. There are many works that have been conducted in this field [SGHP97][ABA+97]. The main known solution is the Real-Time CORBA (RT-CORBA) [SGHP97], which offers a scheduling service based on the timeliness constraints propagation among the nodes connected to the network.

### 4.6.2 Messages characteristics and quality of service

The distributed tasks exchange messages with strict real-time constraints (hard real-time messages). Any failure with the respect of the messages temporal constraints can lead to a breakdown of the service. The respect of these constraints must be guaranteed. On the other hand, messages with relative real-time constraints (soft real-time messages) are characterized by temporal constraints whose occasional non-respect does not lead to the degradation of the service. Finally, non real-time messages may be

exchanged also, these messages do not have any specification of temporal constraints. Consequently, no particular provision is required to consider this type of messages.

Moreover, distributed real-time application requires a subsystem of communication with a certain quality of service which is expressed in particular by the following parameters.

- **Maximum transfer deadline**: it is the maximum time to transmit a message from a node to another. The transfer time of a message is composed of several intermediate times: the time of crossing the protocol stack at the transmitter, the waiting period at the MAC layer of the transmitter before transmission, the message propagation time on the medium, the waiting period at the MAC layer of the receiver and the time of crossing of the protocol stack at the receiver.
- **Maximum jitter**: the messages transmitted by a source can have variable transfer times according to the load of the network. The maximum variation of the transfer time is called the jitter. Certain applications as the multimedia (where the presentation of information requires a synchronisation) specify a boundary for the jitter.
- **Acceptable maximum messages loss**: in case of overloaded situations, the network or I/O driver can then, without informing the user, remove certain messages, in order to serve other sources of messages. For certain applications, like the video or audio applications, the loss of some messages is acceptable. On the other hand, critical systems based on a reactive design and validated by model checking or other formal method are very sensible to this loss since such network behavior invalidates the proof of their model.
- **Error rate**: the communication network being imperfect, errors of transmission can appear at any time. The correct operation of an application can impose a maximum error rate beyond which the application is regarded as seriously compromised. Generally, a probabilistic model is applied here to compute this metric.

After this brief overview on static and dynamic scheduling algorithms and the presentation of some strategies for distributed real-time scheduling, the rest of this chapter will present our contribution to enhance SCOOT-R middleware by distributed real-time scheduling. The main goal is to enforce QoS capabilities to the initial SCOOT-R middleware by the priority, deadline and criticalness propagation for remote operations.

## 4.7 Enhancing SCOOT-R client/server by incorporating distributed scheduling strategies

Initially, SCOOT-R middleware did not include any technique of messages scheduling. The tasks are scheduled by the fixed-priority RTAI scheduler and the management of messages was relatively rudimentary using FIFO scheme. The initial diagram did not take into account the priority of the task sending or receiving the message. The received message was simply piled up in the reception queue. The effective sending of the packet is managed by hardware using OHCI-1394 adapter.

Moreover, original SCOOT-R middleware [CCS03c] did not provide any mechanism for clients to indicate the relative scheduling parameters of their requests to SCOOT-R endsystems[1]. Such a mechanism is necessary for the end-to-end predictability of invocations for distributed real-time applications. Therefore, we get capability from the underlying OS and network bus in terms of CPU utilization and bus bandwidth usage.

[1] A SCOOT-R endsystem includes the middleware layer, the underlying OS and the network adapter.

This section presents our contribution to introduce new specifications and techniques defining the end-to-end timeliness propagation of service invocations. The timing constraints of an invocation might specify the priority, deadline and criticalness of the remote operations.

### 4.7.1 System model and assumptions

Let us consider the distributed system in as a set, N, of n nodes N1, N2,...,Nn, connected by a network. Each node Nj runs a set M of software components M1, M2,..., Mk. Each software component contains several clients and servers and eventually other real-time tasks, Mj=({Sk}, {Cl}, {Tm}). {Sk} represents the set of k servers in the software component, {Cl} the set of l clients and {Tm} is the set of m independent tasks (non SCOOT-R tasks).

Furthermore, tasks are considered as *periodic* or *sporadic*. In order to simplify the following, we do not consider the *interlocked* or imbricated transactions. i.e., when a client *Clm* initiates a transaction to its server *Sm*, the server Sm does not initiate other transactions to other servers in order to achieve its transaction.

Each task Ti is characterized by an activation time Ai, a period Pi (for sporadic tasks, this is the minimum time interval between two activations), a worst case execution time (WCET) Ci, a deadline Di and a criticality Ii.

Ai is the activation instant of the task Ti. We assume in our model that the tasks are *sporadic*. The worst case execution time Ci (assumed known *a priori*) is the longest CPU time required to execute a single occurrence of a task.

The deadline Di denotes the deadline of the task Ti. It is assumed to be known *a priori* and it is obvious that Di $\geq$ Ci. Di is relative to the Ti activation time Ai.

The task criticalness Ii is an application-supplied value that indicates the significance of a task's completion prior to its deadline. Higher criticality should be assigned to tasks that incur greater cost to an application if they fail to complete execution before their deadlines. Some scheduling algorithms, such as MUF (§4.5.3), give greater priority to more critical tasks than to less critical ones. The tasks are assumed to be preemptable with fixed-priority based scheme. A task execution may be preempted by higher priority task activation. In our proposed scheduling strategies, CDP-BBA (§4.7.2.3) is the only scheduling strategy that uses the criticalness and the laxity scheduling parameters.

A transaction message is assumed to be transmitted in one packet by the sender task. A transaction M can be associated to a priority *Mp*, an absolute deadline *Md*, and a criticality *Mi*.

Priority, deadline and criticality of the transactions are also supplied by the applications. The sender task priority Pi will be transmitted to the corresponding priority transaction *Mp*. The transaction deadline *Md* is the one specified in the SCOOT-R contract (client side). The transaction criticality *Mi* is supplied for this purpose by the client application.

Similarly to the task model, we use *Mr(t)* to denote the remaining execution time of a transaction M at time t. As the communication model in our study is based on a client/server model, we define the transaction execution time as the amount of time for the client to get the reply back. Thus, it is equivalent to the server's task computation time plus the communication delays and overhead of network adapters.

We define the laxity of the transaction M at time t as:

$$Ml\,(t) = Md\,(t) - t - Mr\,(t) \tag{4.7}$$

Each transaction is assigned a fixed priority. Packets are queued in FIFO order queue shared between the host processor and the network adapter. The packets are queued by the sender task.

Let's recall that the server processes clients requests so that the arrival of a new client request will not preempt the current server request processing even if its priority is greater than the current request priority being processed by the server.

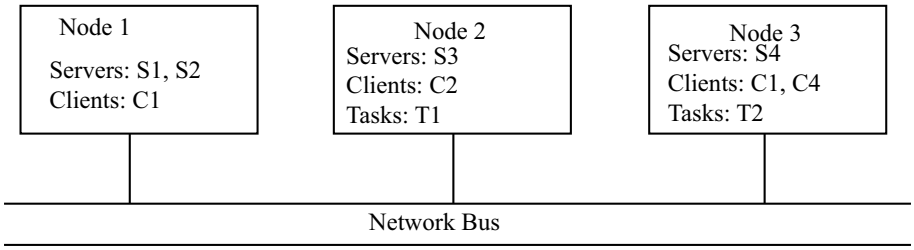



Figure 4.4: Example of a system configuration

Moreover, due to the distributed nature of the system, even if the initial data are sampled periodically, the release jitter in the later stages of processing becomes so large that the strict periodic task model does not apply there.

In addition, as the distributed system is based on client/server model, exchanged messages are closely linked to the temporal specification of the real-time clauses of the client/server model .

### 4.7.2 Integrated messages and tasks scheduling

The target application of our scheduling strategies is a set of end-to-end *sporadic* tasks. Each *sporadic* task is triggered by the arrival of a sensor data message, a timer event, or data message arrival from the network. Part of these tasks will initiate SCOOT-R transactions.

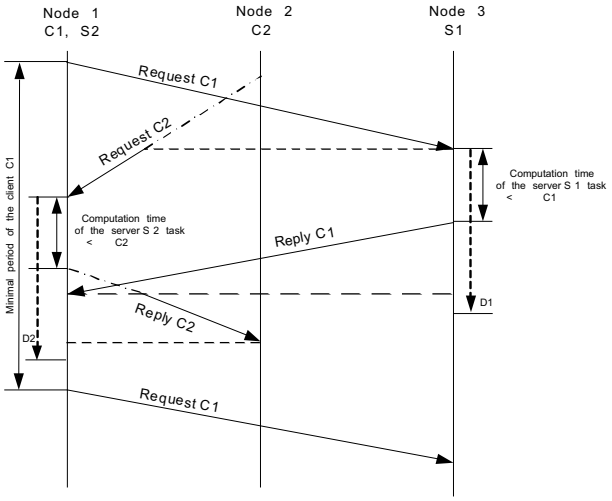



Figure 4.5: Media and CPU sharing example

As shown in Figure 4.5, tasks and messages scheduling are strongly dependent. The client C1 located on node1 initiates its request and gets the media communication until its request has been received by its Server S1 located on node3. Therefore, the request of the client C2 located on node2 will be transmitted after the request of C1.

#### 4.7.2.1 Client Priority Propagation (CPP)

**Description:** This model allows clients to declare their transactions priority that must be honored by their servers. In this model, each client request message carries the client task priority.

The *middleware bus* carries this end-to-end client task priority along the communication path to the server. This priority will be used to order (ascending priority-based order) the messages at the emission of clients requests and servers replies on the network. In case of several clients of one server in the system, all the clients requests for this server are scheduled in prioritized queue at the server side. So, the client having the highest priority will be served first. Consequently, the client task priority in the client request will determine the priority of its reply (Figure 4.6).

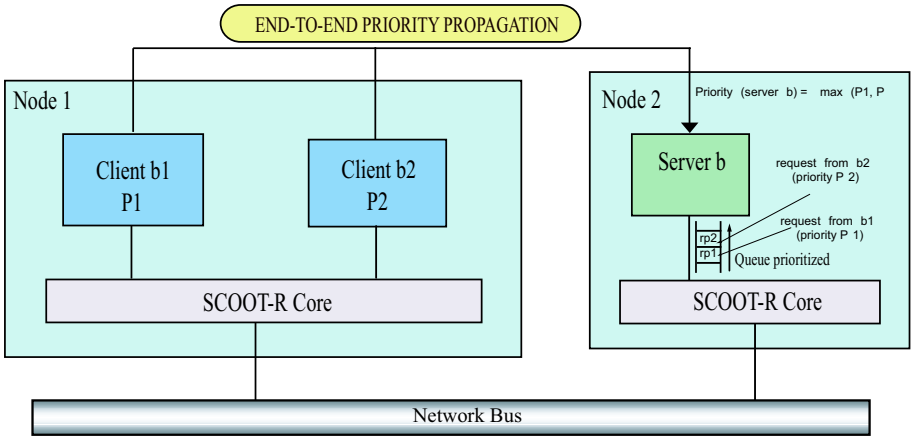



Figure 4.6: CPP: Client Priority Propagation

In order to avoid priority inversion at the server node, the server's task processes incoming requests at a priority equivalent to the maximum priority of all its current clients requests (presented in the prioritized queue). For example, as shown in Figure 4.6 the server "b" has two requests "rp1" and "rp2" from two clients "b1" and "b2", the server "b" will have the maximum priority of b1 and b2 (Max(P1, P2)).

Recall that, in initial SCOOT-R, incoming messages are processed using FIFO queues in accordance with the sequence in which messages are generated. If messages are processed in the queuing order, the processing of high priority message can, therefore, be delayed by low priority messages that entered earlier. As a result, the message-processing time is not determined by the priority of the messages waiting to be processed, but rather by the number of messages that are waiting in the FIFO queue. The message processing jitter time also changes relatively to the network overhead. To solve these problems, messages must be processed in order of priority using priority-based queues. The tasks that process messages must be able to be preempted by tasks dealing with high-priority messages and scheduling together with the user tasks.

**Implementation of CPP in SCOOT-R:** In order to implement the CPP scheduling technique in SCOOT-R, the message header for clients requests was increased to include the priority of the client task (Figure 4.7).

The incoming requests messages at the server side are queued in a prioritized queue according to their priorities. We have one queue by server (Figure 4.8).

SCOOT-R's CPP scheduler takes in charge the modification of the server's task priority according

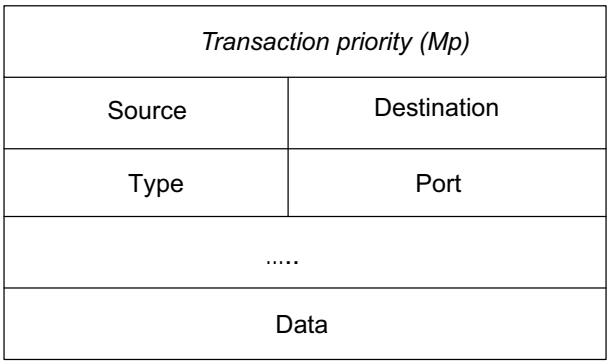



Figure 4.7: CPP implementation

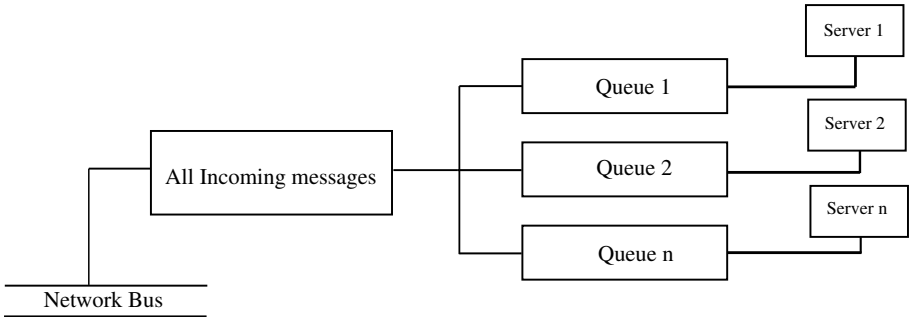



Figure 4.8: Queues Architecture

to the priorities of its current clients requests. Let's note that sorting the messages at the emission of clients requests and servers replies is not currently implemented in CPP. We use the facility of the OHCI-1394 adapter that manage automatically the messages lists for emission and reception.

**Evaluation of CPP:** The Client Priority Propagation (CPP) scheme is relevant in case of distributed applications with fixed-priority profile. It permits to reduce the end-to-end priority inversion, as well as to bound latency and jitter for invocations with higher priorities. Like the RM algorithm (§4.4), CPP provides schedulability *assurance* prior to run-time for invocations with higher priorities and in case of overloaded situations. On the other hand, CPP offers an equivalent performance for lower priorities invocations. The client propagation model is used in the real-time implementation of CORBA ((TAO/ACE).

From the methodology point of view, CPP imposes a global RM analysis for the tasks set of all the components in the distributed architecture. This approach is not compliant with the philosophy of the component approach and is neither adapted to the COTS-based development.

To overcome the limitations of static priorities propagation, therefore, we have envisaged the development of deadline propagation scheme outlined below.

#### 4.7.2.2 Client Deadline Propagation (CDP)

**Description:** CDP allows clients to declare their requests deadlines that must be honored by their servers. In this model, each client request message carries the client task *deadline*.

The *middleware bus* carries this end-to-end client task deadline along the communication path to

the server. This deadline will be used also to order (ascending deadline-based order) the messages at the emission of clients requests and servers replies on the network bus. In case of having several clients for one server in the system, all the clients requests for this server are scheduled in deadline-based ordered queue at the server node. So, the client having the closest deadline will be served first (Figure 4.9).

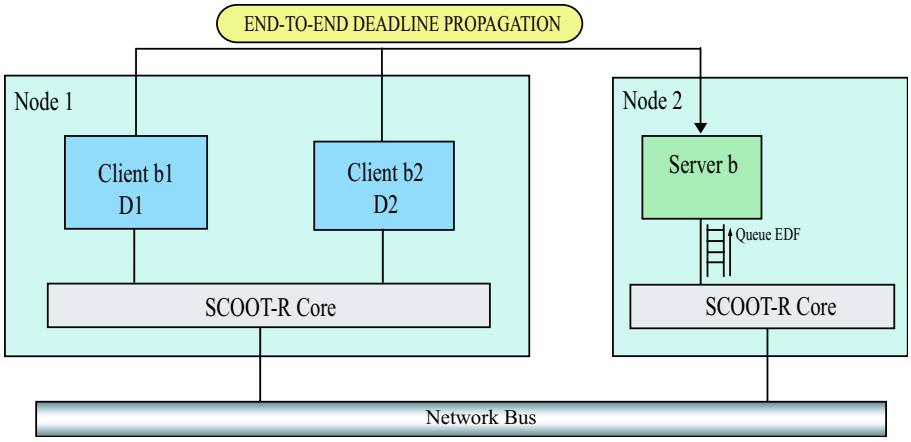



Figure 4.9: CDP: Client Deadline Propagation

The deadline parameter specified in the clients requests has an *absolute* value (absolute deadline). Thus, CDP model cannot be implemented without having a global time base on all the nodes. Using *absolute deadline* instead of *relative deadline* allows the implementation of an end-to-end distributed EDF scheduling strategy. As shown in Figure 4.10, and in case of *relative* deadline propagation (TAO CORBA), a transaction t1 having an earliest deadline than the transaction t2 ($w(t1) < w(t2)$) but has arrived late at the node3 ($e(t2) < e(t1)$), will be processed after the transaction t2 by the server on node3. The propagation of *absolute* deadline will permit to absorb this late arrival of requests and thus a global and coherent scheduling decision may be taken (end-to-end distributed EDF).

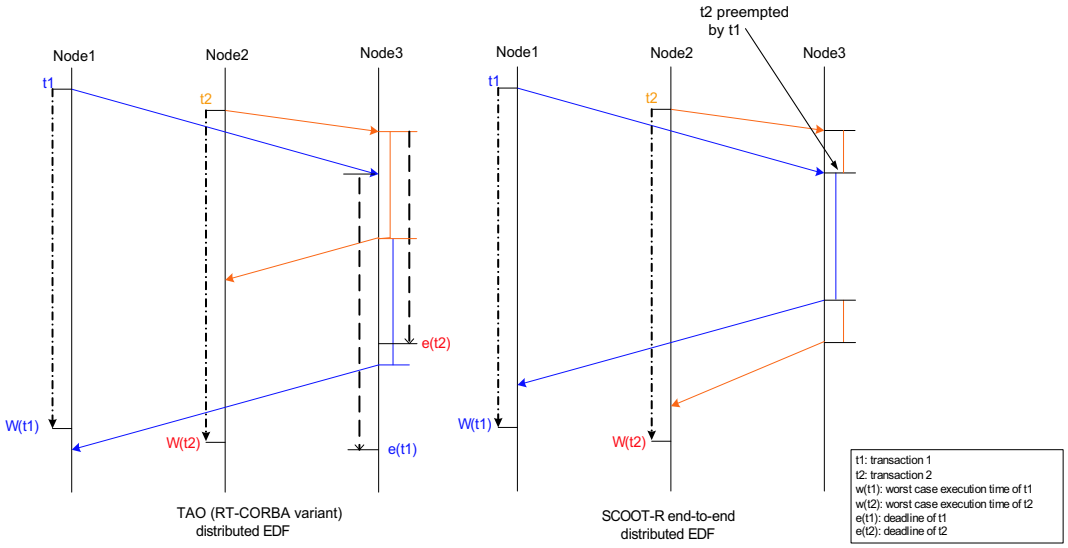



Figure 4.10: Distributed vs. end-to-end distributed EDF

As outlined in Algorithm 4.1, only messages that have their deadlines greater than the current time

can be in the feasible schedule. Thus, CDP discards all requests with missed deadlines.

**Algorithm 4.1** CDP algorithm

*Input : Absolute deadline ordered messages (requests) ready queue DQ;*

*For each message M in DQ in ascending order of Md*
$t = get\_current\_time()$;
**if** $Md(t) \geq t$ **then**
//We schedule only the messages that have deadlines > current time
*Execute M*; //Here we execute the message M until its end
**end if**

CDP has as input the DQ queue ready requests (deadlines-based ordered). The algorithm examines the messages in the ready queue in ascending order of their deadlines. Note that the algorithm schedules requests at two scheduling events: 1) the arrival of a new request and 2) the termination of the currently executing request.

The number of requests in the queue $DQ$ that have longer deadlines than $di$ may be approximated by:

$$k = |DQ| \times \frac{d\,max - d\,i}{d\,max - d\,min} \tag{4.8}$$

where *dmax* and *dmin* are the maximum and minimum deadlines among all requests currently in DQ, respectively. Thus, k is the number of requests that will be affected by inserting *Mi*.

Moreover, the local servers on a node are scheduled using an EDF online scheduler. The deadline of the server's task will be the minimum deadline between the request being processed and the request at the top of the ready requests queue DQ (lowest deadline).

**Implementation of CDP in SCOOT-R:** In order to implement the CDP scheduling strategy in SCOOT-R, we have augmented the message header by the deadline of the transaction (Figure 4.11). A message arriving at the server node causes a message arriving event. The CDP scheduler extracts the *transaction deadline* value of the client request, and then it orders them in ascending deadline-ordered queue to be proceeded by the server.

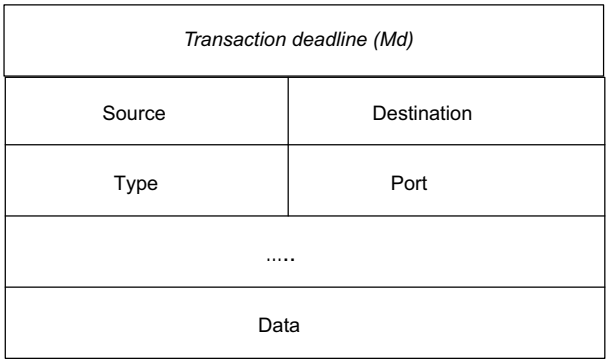



Figure 4.11: CDP implementation

In CDP SCOOT-R implementation, we associate to each server a reception queue (DQ). The clients requests are queued into this reception queue (DQ) according to their deadlines (ascending deadlines order). The CDP scheduler then uses the deadline information in order to select the client request to be proceeded. The request with the earliest absolute deadline will be executed first (Figure 4.12).

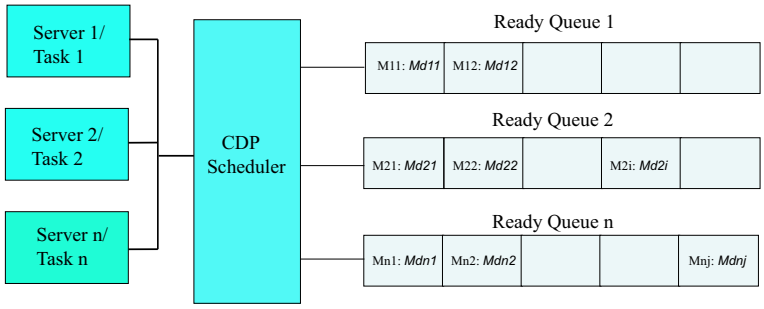



Figure 4.12: Deadline-based scheduling

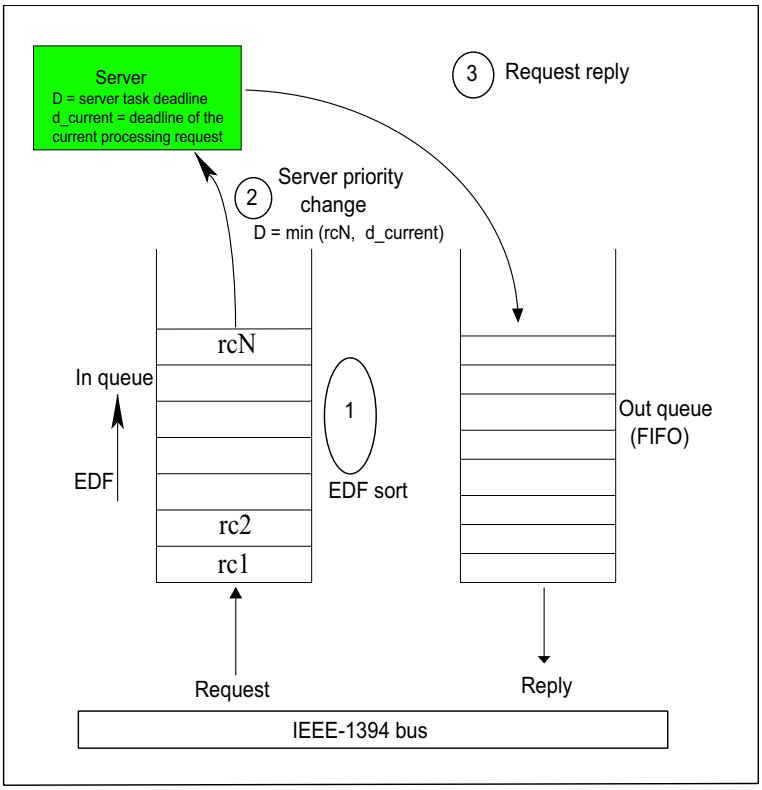



Figure 4.13: Tasks and messages scheduling

In order to obtain absolute deadline of distributed clients transactions we use the service provided by the SCOOT-R middleware that permits to obtain a global time base of the distributed system (§3.6.1).

In RTAI native kernel, the tasks priorities are encoded on 32 bits. Thus, we have modified the kernel to encode the priority on 64 bits. So, setting the task priorities to the deadlines implements directly an EDF scheduler without the limitation of the RTAI EDF's mode.

Using our EDF scheduler, the priority of the server's task will be the minimum of the current processing request deadline and the deadline of the request at the top of the *in queue* (EDF ordered) (as depicted in Figure 4.13). Moreover, the *out queue* in Figure 4.13 is a FIFO-based queue since we did not implement an ordered emission of messages on the IEEE-1394 bus. As the CPP implementation, we use the facility of the OHCI adapter that manages automatically the emission/reception of messages.

**Evaluation of CDP:** While the *client priority propagation* model is useful for transactions in fixed-priority applications, the Client Deadline Propagation (CDP) model reflects perfectly the timing profile of distributed transactions. The client request deadline gives a global and dynamic interpretation of the timing constraints. Clients may express their timing profile by an arbitrary deadline and thus a coherent scheduling may be implemented on all the nodes.

From a methodology point of view, CDP is well adapted to our middleware proposition SCOOT-R and more widely to any component-oriented approach. Moreover, CDP is adapted to mixed networks when several nodes apply the EDF policy and other limited nodes (e.g., microcontrollers) work with the FIFO scheme.

Unfortunately, the weakness of the CDP model is the performance degradation in case of overloaded situations. Like all the purely dynamic scheduling algorithms (e.g., EDF, MLF), a transient overload in the system may cause a critical task to fail, which is not desirable for a dynamically reconfigurable system. So, the decision about the criticalness of messages/tasks must be taken in run-time. Next section presents a hybrid static/dynamic scheduling technique that performs in overloaded situations. It allows to schedule the tasks/messages while taking into account the criticalness at run-time.

#### 4.7.2.3 CDP-BBA scheduling strategy

To alleviate the limitations with the CPP and CDP scheduling policies in overloaded situations, many research works have been conducted in the field of overload real-time scheduling. The main objective is to offer control over which invocations will be sacrificed under overload conditions. Below, we present some implementations of overload real-time scheduling for mono-processor architecture.

Two recent overload and best-effort real-time scheduling are the Dependent Activity Scheduling Algorithm (DASA) and Locke's Best Effort Scheduling Algorithm (LBESA) [LRWK04]. DASA and LBESA are equivalent to the Earliest Deadline First (EDF) algorithm during underloaded conditions. In overload situation, DASA and LBESA permit to maximize the overall tasks benefit by guaranteeing a schedulability of the critical part of the system. i.e., they associate to each task of the system a *benefit* value, thus maximizing the *overall tasks benefit* returns by maximizing the sum of benefit values of the tasks set.

Li and al developed in [LRWK04] two fast best effort algorithms MLBESA and MDASA that outperform LBESA and DASA algorithms in terms of speed. They use heuristic model and methods to decide about scheduling of tasks. They consider an asynchronous environment and the target tasks of their algorithms are aperiodic and independent. MLBESA and MDASA algorithms may be directly adapted and integrated to our SCOOT-R middleware in order to schedule integrated tasks/messages in a distributed environment.

Another implementation of overload real-time scheduling is the MUF (Maximum Urgency First) algorithm (§4.5.3). MUF combines the advantages of *static* and *dynamic* scheduling techniques in order to deal with transient overload situations. DASA, LBESA and their extensions may be considered as variants of the MUF scheduling strategy.

Hereafter, we present our CDP-BBA[2] scheduling strategy. CDP-BBA is similar to MUF strategy applied to the scheduling of messages and tasks in a distributed environment and using the middleware approach (client/server model).

CDP-BBA overcomes the drawbacks of the individual scheduling strategies (CPP and CDP) by combining techniques from each as outlined below.

[2]CDP-BBA: Client Deadline Propagation and Best Benefit Algorithm

**Description:** CDP-BBA schedules arriving messages and associated tasks based on both the native *criticalness* and the *absolute deadline* of the arriving messages (Figure 4.14). In CDB-BBA, transactions with higher criticalness are assigned to higher static priority levels.

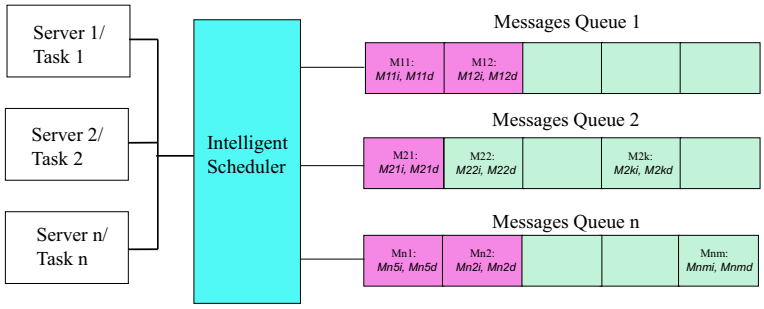



Figure 4.14: CDP-BBA scheduling strategy

We denote the criticalness of a message M at time t as *Mi* (§4.7.1). As shown in Algorithm 4.2, CDP-BBA takes as input a messages ready queue IDQ. IDQ queue orders the incoming messages in ascending criticalness (criticalness-order) and then in ascending deadline. Thus, if two or more messages have the same criticalness value, the message with the earliest deadline will be inserted before. Let's recall that $M1i < M2i$ means that the message M1 is more critical than the message M2.

**Algorithm 4.2** CDP-BBA scheduling algorithm

*Input* : *Criticalness and Deadline ordered messages ready queue IDQ*;
*Queue* $\Gamma$ *of messages to be executed*
*Initialize* : $\Gamma \leftarrow \phi, C \leftarrow 0$;
*for each message M in IDQ in ascending order of Mi and then of Md do*
**if** $Ml(t) \geq 0$ **then**
  **if** $\Gamma = \phi$ **then**
    $\Gamma \leftarrow \Gamma \cup M$;
  **end if**
  **if** $\frac{C+Mr(t)}{Md-t} \leq 1.0$ **then**
    $\Gamma = \Gamma \cup M; C \leftarrow C + Mr(t)$;
  **end if**
**end if**

CDP-BBA schedules messages at two scheduling events: 1) the arrival of a new message and 2) the termination of the currently executing message.

**Implementation of CDB-BBA in SCOOT-R:** In order to implement CDP-BBA algorithm in our SCOOT-R middleware, we have augmented the message header by the criticalness of the transaction (Figure 4.15).

This criticalness indicator *Mi* encoded on 32 bits is combined with the deadline parameter (64 bits) *Md* to choice the higher priority transaction.

The local scheduling of servers follows the same scheme that of MUF. Thus, the server's task priority inherits from the client request criticalness *Mi*. If two requests have the same criticalness, the scheduler assigns to the server task the priority that corresponds to the request deadline *Md*.

**Evaluation of CDB-BBA:** With CDP-BBA, all critical messages are guaranteed not to miss deadlines as long as there is sufficient resources available. Thus, CDP-BBA provides a higher scheduable bound of the critical tasks and messages. Unlike CDP, where any message may have failed as there is no way to predict, CDP-BBA permits to control which messages may fail.

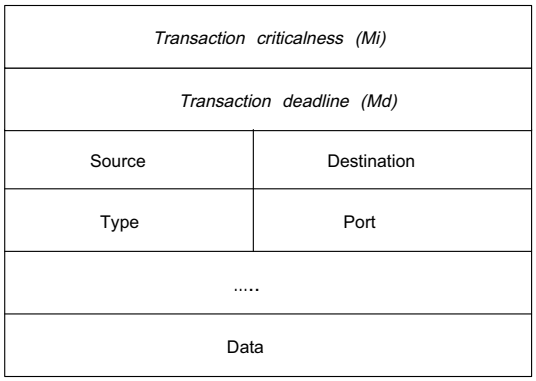



Figure 4.15: Best benefit algorithm implementation

By assigning dynamic subpriority according to the transaction deadline, CDP-BBA offers a higher utilization of the CPU resources and gives a coherent representation of scheduling decisions when the transactions having the same criticalness value.

CDP-BBA is well suited for dynamically reconfigurable systems where the global state of the system may change without a system halt. In such system, it is crucial that critical components do not fail. By allowing applications to select which operations are critical, it is possible to provide scheduling behavior that is appropriate to each application's individual requirements. Thus, if it is possible to ensure that missed deadlines will be isolated from non-critical operations, then adding non-critical operations to the schedule to increase CPU usage will not increase the risk of missing critical deadlines. The risk will only concern those operations whose execution prior to the deadline is not critical to the integrity of the system.

Note that using such scheduling strategy with SCOOT-R will need some adaptation for the *server unregistration* initiated by the clients. The simplest solution is to define a *server criticality* so only the client with criticality higher than the server one can unregister it. It is envisaged also to explore more dynamic solutions with more flexibility.

Finally, the overload scheduling algorithms DASA and LBESA and their variants MLBESA and MDASA use the same scheduling data and need similar propagation mechanism as the MUF strategy. They may be easily adapted to the distributed environment using the middleware client/server methodology the same way our CDP-BBA derives from MUF.

## 4.8 Performance evaluation and experimental results

In order to evaluate the performance of the proposed distributed scheduling strategies and validate the implementation approach, we have built a real experimental testbed based on our SCOOT-R middleware. It is composed of several computers connected by an IEEE-1394 bus. This section begins by the description of the testbed architecture, then the experimental results obtained are presented and discussed. Finally, we will present the simulation results using the same testbed architecture and built-up using the Matlab toolbox "truetime" [HCr03].

### 4.8.1 Testbed architecture

Our experimental testbed comprises four Pentium-based computers (800 MHz) connected by an IEEE-1394 (400 Mb/s). Each computer has SCOOT-R middleware installed to implement our applications.

The underlying OS is the real-time Linux microkernel (RTAI).

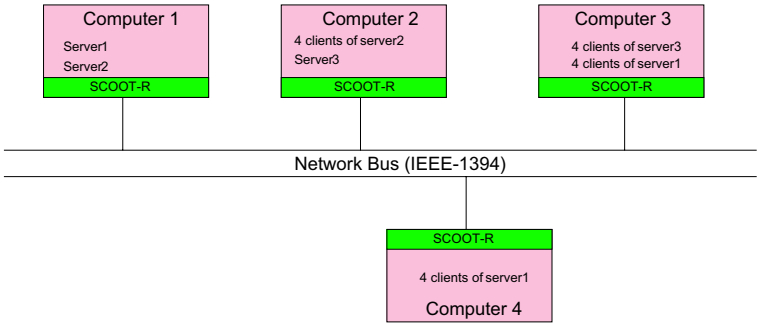



Figure 4.16: Application configuration

As depicted in Figure 4.16, each computer of the testbed contains several clients and servers. Table 4.1 summarizes the baseline experimental settings for clients and servers in each computer of the Figure 4.16.

| Components / Computer | Clients | Servers |
|---|---|---|
| Computer 1 | No clients | server1: C1 = 2.5 ms; server2: C2 = 1 ms |
| Computer 2 | client*22i*: D22i = 40 ms, i =0 to 3 | server3: C3 = 1 ms |
| Computer 3 | client*33i*: D33i = 30 ms + i*5 ms, i = 0 to 3; client*31i*: D31i = 30 ms + i*4 ms, i = 0 to 3 | No servers |
| Computer 4 | client*41i*: D41i = 15 ms + i*5 ms, i = 0 to 3 | No servers |

Table 4.1: Baseline experimental settings

Recall that client<*m*><*n*><*i*> denotes the client on computer m, client for server n, and *i* is the index of this client (in case of multiple clients for the same server). D<*m*><*n*><*i*> denotes the *relative* worst case deadline for client<*m*><*n*><*i*>. For example, D*331* = 30 ms + 5 ms = 35 ms and D*413* = 15 ms + 3*5 = 30 ms. To obtain absolute deadlines of clients transactions, it's sufficient to add D<*m*><*n*><*i*> to the current common time base T*common(t)*.

Ci denotes the worst case server computation time. We assume in this experimentation that the *server computation time = worst case server execution time (C) = server deadline (D)* and remains constant for all the server requests.

In order to bursty load the server, the clients requests have the same period and they start at the same common time with a small random shift (Figure 4.17).

Let's recall that this testbed is useless for real applications and it is designed just to verify the relevance of the simulation model of our scheduling strategies. Thereafter, we present the experimental and simulation results obtained using the *CDP* scheduling strategy and compared to the FIFO scheme.

### 4.8.2 Experimental results

By experimentally evaluating CDP scheme, our main goal is to determine how well this strategy performs in *bursty workload* situations with respect to applications temporal requirements.

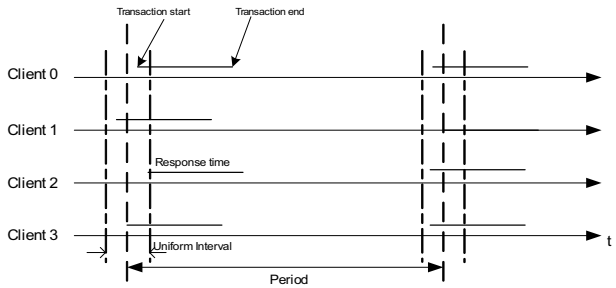



Figure 4.17: Random shift of transactions sequence

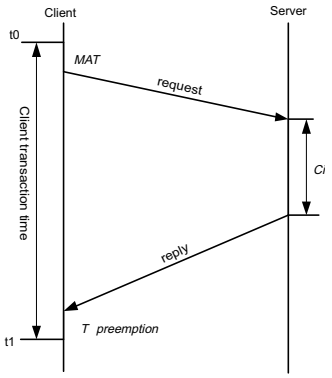



Figure 4.18: Client/Server transaction time

A client/server transaction time is depicted in Figure 4.18. The transaction time is composed of several elementary times. MAT (Media Access Time) corresponds to the time that the client has to wait before it accesses the bus and begins the transmission. $Ci$ is the worst computation time that the server's tasks consumes to process a client request. $T$ *preemption*, is the time between the arrival of the reply on the client node and the effective reception of the reply by the client (this delay is due to the overhead of network adapter, OS software and SCOOT-R layers).

In order to evaluate the performance of the CDP scheduling technique, the experimentation has been conducted using the testbed architecture outlined before (§4.8.1).

Each experiment test lasted for 10 hours. Figure 4.19 shows the responses time of the four clients located on computer4 (client$41i$) in the case of *deadline propagation* scheme (CDP). The clients requests are assumed to an average period of *100 ms*. We illustrates in Figure 4.19 only a limited number of transaction to well show the behavior of the clients transactions.

As depicted in Figure 4.19, the client$410$ response times vary around 2.78 ms. The peak in the client$410$ response time (around the transaction number 2480) corresponds to the fact that the request of the client$410$ arrives when the server1 is processing a request from the client$411$.

Client$411$ has approximatively 5.43 ms of average response time. The peak around the transaction number 2390 corresponds to the delay implied by the processing by the server of a request of the client$410$ and a previous request from another client. At the transaction 2480 approximately, the client$411$ request has arrived and processed before the client$410$ request.

The client$412$ and client$413$ have an average response times that vary around 8 ms for client$412$ and 10 ms for client$413$. Client$413$ has the worst average response time and that corresponds to the worst server1 load. i.e., when a client$413$ request arrives to the server1, it has in its list three requests

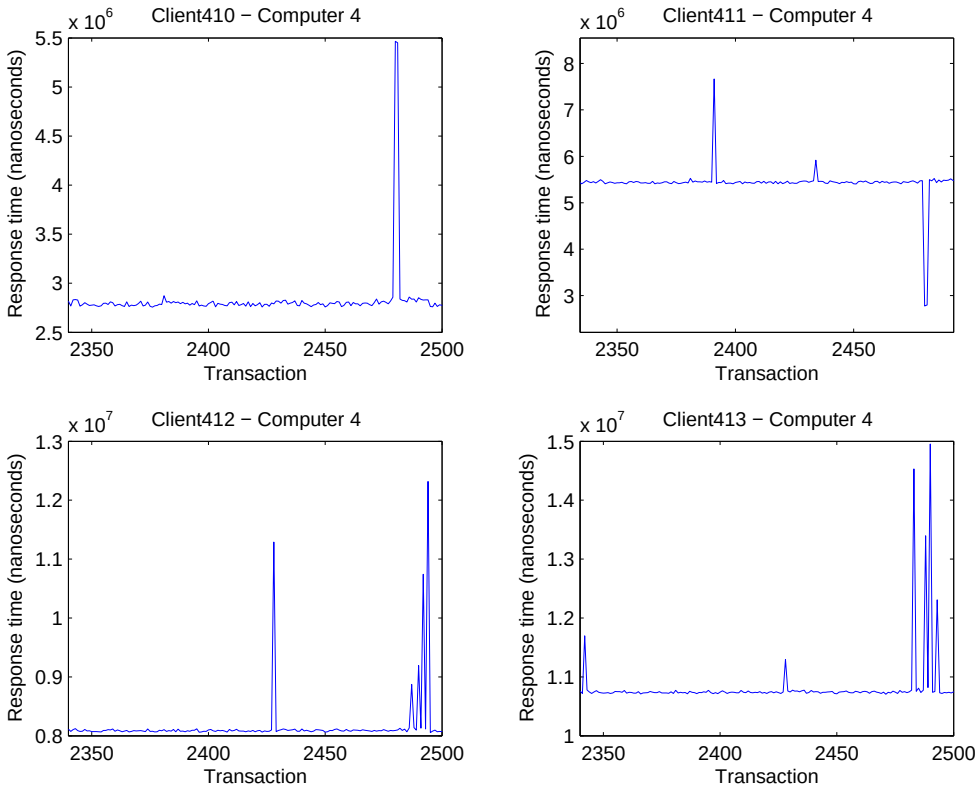



Figure 4.19: Clients response times with CDP

from clients having higher priority (client*410* to client*412*). Thus, the server1 will process the requests in deadline-based order, so the client*413* will be served last.

In Figure 4.20, we illustrate the transactions response time in case of non transmission of the clients requests deadlines (FIFO transmission scheme). As shown, the four clients (client*410* to client*413*) have the same behavior and thus the same performances.

### 4.8.3 Simulation results

In this section, we present some simulation results for the proposed scheduling techniques. The simulation developed here was mainly motivated by the validation of the testbed architecture described earlier in order to evaluate our proposed feedback scheduling depicted in the next chapter. In addition, this simulation permits to validate some possible configurations that are hardly implemented with real experimentation. This simulation has allowed the validation of the SCOOT-R model to design and implement end-to-end scheduling strategies.

For this purpose, we have used Truetime Matlab toolbox that allows the simulation of distributed real-time control systems. Truetime makes it possible to simulate the timely behavior of real-time kernels executing control tasks. Truetime simulates simple models of network protocols and their influence on networked control loops.

Truetime consists of a kernel block and a network block, both variable-step S-functions written in C++. It also provides a collection of Matlab functions used to do A/D and D/A conversion, send and receive network messages, set up timers, and change task attributes. The Truetime blocks are connected with ordinary continuous Simulink blocks to form a real-time control system.

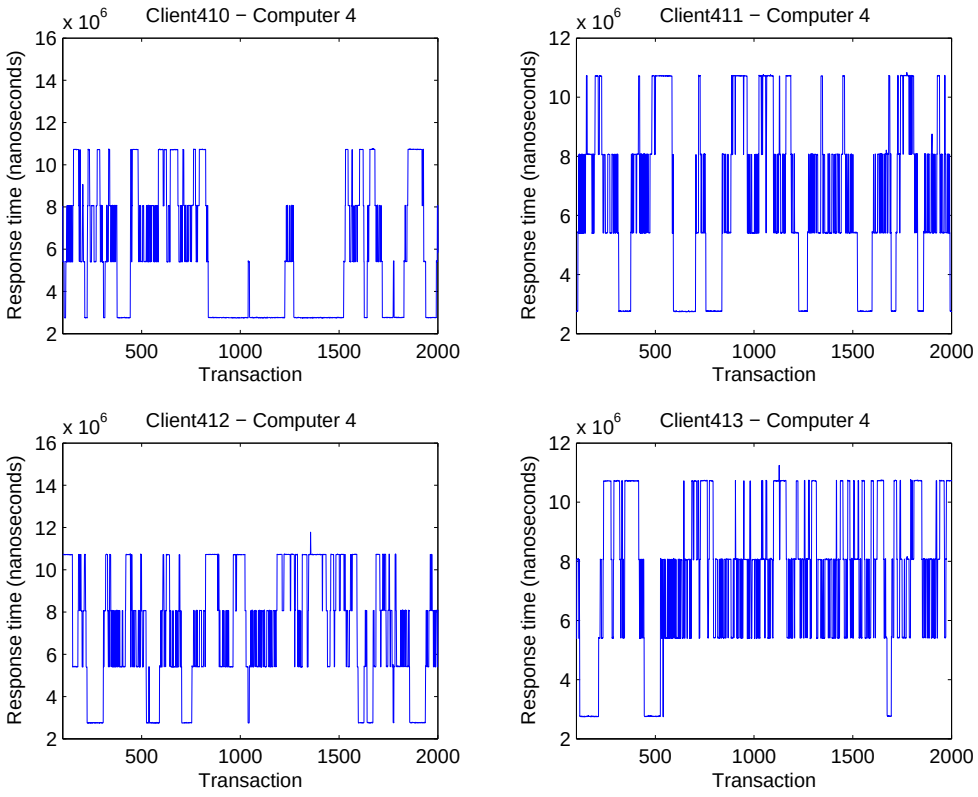



Figure 4.20: Clients response times with FIFO scheme

The Truetime kernel block simulates a computer with an event-driven real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels.The kernel executes user-defined tasks and interrupt handlers, representing, e.g., I/O tasks, control algorithms, and communication tasks. Execution is defined by user-written code functions (C++ functions or m-files) or graphically using ordinary discrete Simulink blocks. The simulated execution time of the code may be modeled as constant, random or even data-dependent. Furthermore, the real-time scheduling policy of the kernel is arbitrary and decided by the user.

The Truetime network block is event driven and distributes messages between computer nodes according to a chosen network model. Currently five of the most common medium access control protocols are supported: CSMA/CD (Ethernet), CSMA/CA (CAN), token-ring, FDMA, and TDMA. It is also possible to specify network parameters such as transmission rate, pre- and post-processing delays, frame overhead, and loss probability.

Using TrueTime, we have implemented the same testbed architecture described in section 4.8.1. Thus, we have simulated the four computers of our experience by computer blocks and the IEEE-1394 network by a CSMA/CD Ethernet network block (Figure 4.21).

Each server and client in the software architecture are represented by *kernel tasks*. The client/server communication is performed using the notion of *port*. A port number is associated to each client (in case of multiple clients for the same server), this port number will be used by the ISR (Interrupt Service Routine) to dispatch the transactions to the appropriate clients. The scheduling parameters such as the priority, deadline and criticalness are encapsulated in the header of each exchanged message. We consider that all the clients have the same period with a voluntary shift to overload the server (normal random distribution).

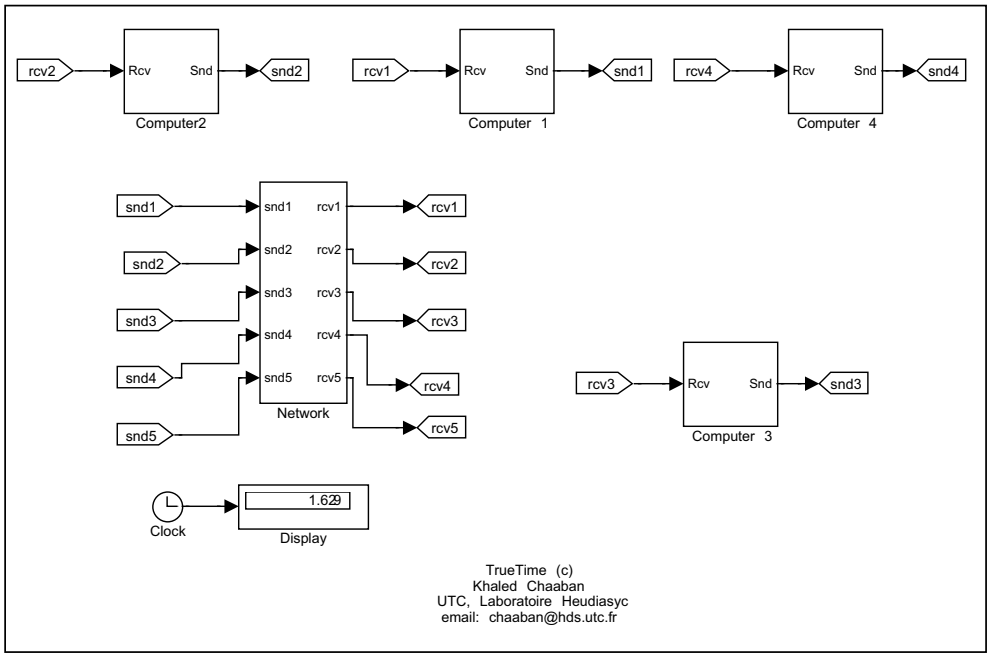



Figure 4.21: Our application architecture modeled by TrueTime toolbox

The server, implemented as a non-periodic kernel task, processes clients requests using a reception queue. A reception of a request is handled by a specified event that the server monitors (Figure 4.22). The local scheduling policy used is the Fixed-Priority scheduling. The implementation of the CPP

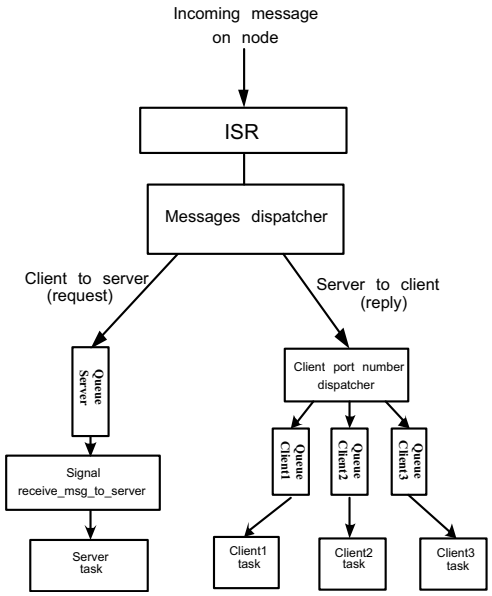



Figure 4.22: Dispatching incoming messages to clients and servers

and CDP algorithms consist of the replacement of the FIFO incoming queue at the server side by a prioritized one.

The simulation results are similar to those of the real experimentation. Figure 4.23 shows the transaction response time in seconds for the four clients (client*330* to client*333*) and using the CDP

scheduling strategy. The X axis represents the transaction number sequence and the Y axis the transaction response time in seconds. We consider that the server3 has a computation time $C3 = 1\ ms$.

As shown in Figure 4.23, the response time of the client*330* is worth 1 ms or 2 ms. This is related to the case where the client*330* request arrives the first to the server3 or it arrives when the server3 is processing another request from another clients request. Let's note that the client*330* has the highest priority (closest deadline) followed by client*331*, client*332* and client*333* (highest deadline).

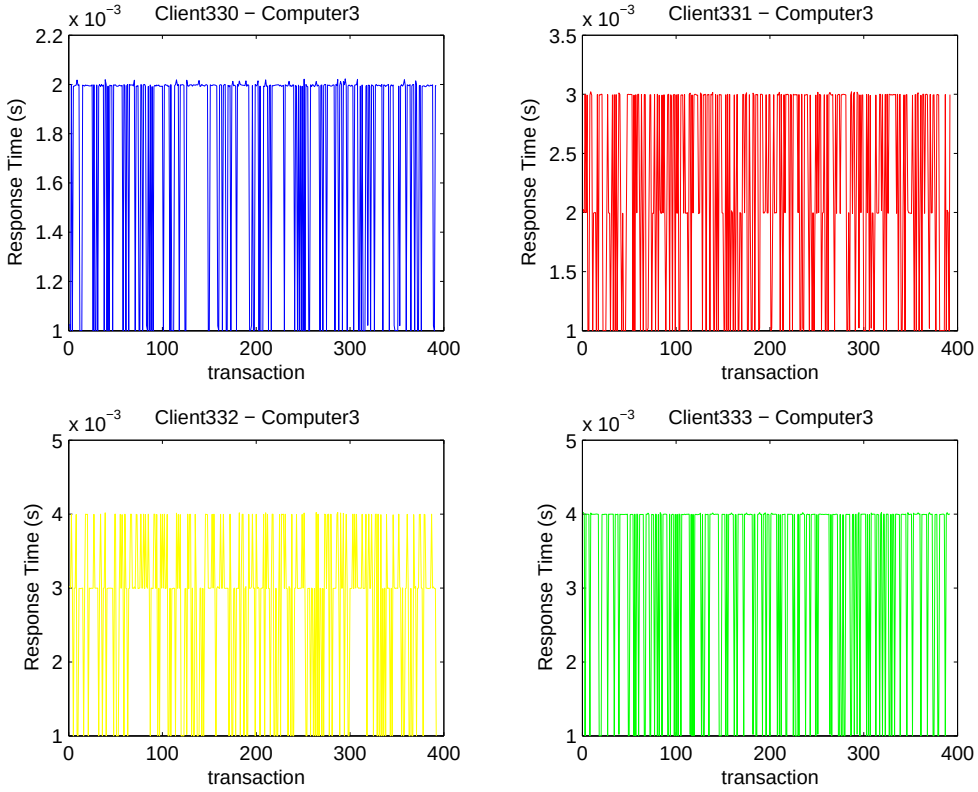



Figure 4.23: With CDP scheme

The response time of the client*331* varies between 1 ms , 2 ms and 3 ms, depending on the number of requests to be processed by the server3 at the arriving instant of the client*331* request.

The response time of the client*333* switches between 1 ms and 4 ms. This means that the client*333* request may arrive the first to the server3 (1 ms) or not and so processed at the end of all the server3 requests (4 ms).

To evaluate the CDP scheduling technique in case of *overload* situations, we have conducted a simulation with a variable workload. In order to modify the *average load*, we have added a *periodic* interfering task with constant CPU usage to the computer including the server. To load the CPU, we increase the interfering task period regularly each 2 seconds of the total simulation time.

We define also the "loss ratio" as the ratio of missed transactions to the total transactions. Figure 4.24 shows the "loss ratio" of four clients with different deadline parameter (client*330* to client*333*).

Client*330* with the highest deadline has the lowest "loss ratio". Client*333* with the lowest deadline has the highest "loss ratio". Let's note also that the "loss ratio" of client*333* becomes 1 rapidly with the average load of the system. Client*330* has always the lowest "loss ratio" whatever the average load

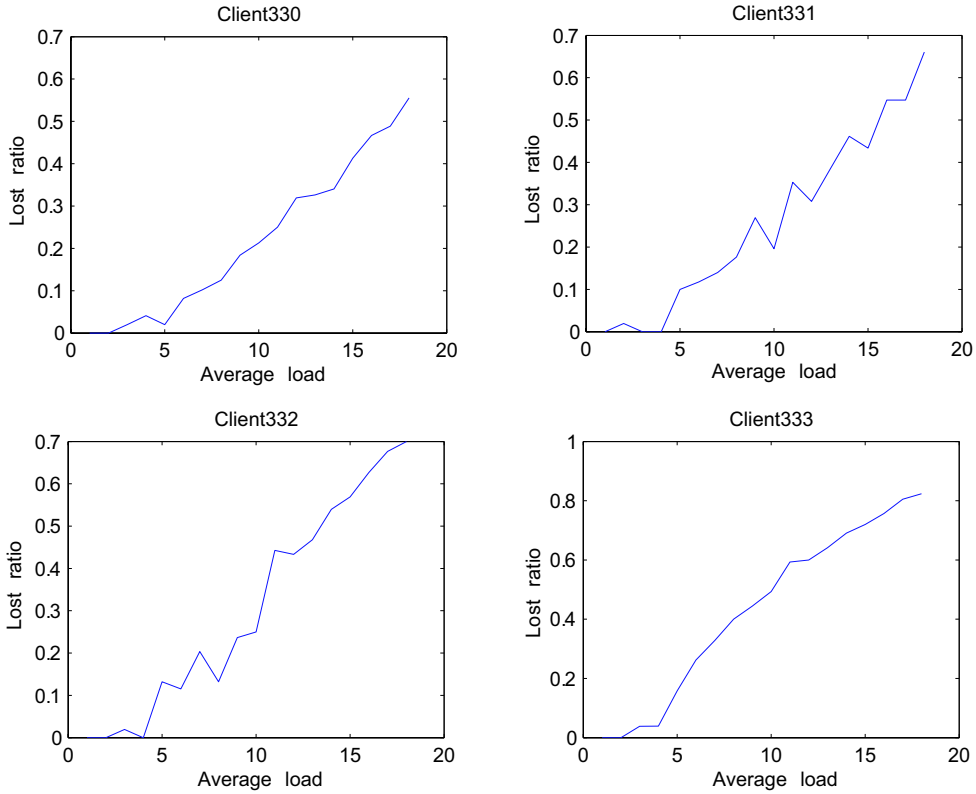



Figure 4.24: Using CDP algorithm

is.

## 4.9 Conclusion

Middleware in a distributed real-time system can provide the platform with mechanisms required to perform the necessary local/distributed scheduling. Our distributed system is composed of nodes interconnected by a deterministic network bus. The target applications are qualified by their timing constraints. These applications may include functions with different levels of criticalness. Moreover, our proposed scheduling techniques are well adapted for a transactional models and are implemented using our SCOOT-R middleware.

Two distributed scheduling strategies were developed, (1) the Client Priority Propagation (CPP), and (2) the Client Deadline Propagation (CDP). We have compared these scheduling strategies with the trivial FIFO scheme.

The CPP is relevant in case of distributed application with fixed-priority profile. During overload situations, it provides a performance benefit for higher priorities invocations but an equivalent performance for lower priorities invocations.

The CDP strategy is equivalent to the Earliest Deadline First (EDF) scheduling strategy. CDP gives a global and dynamic interpretation of the timing constraints while making decision rules. Clients may express their timing profile by an arbitrary deadline and thus a coherent scheduling may be implemented on all the nodes.

From the methodology point of view, CDP is well adapted to our SCOOT-R middleware and

more widely to any component-oriented approach. Moreover, CDP is adapted to mixed networks when several nodes apply the CDP policy and other limited nodes (e.g., microcontrollers) work with the FIFO scheme.

Unfortunately, a transient overload in the system may cause a critical task to fail, which is not desirable for a dynamically reconfigurable system. So, the decision about the criticalness of messages/tasks must be taken in run-time.

Thus, to deal with such overloaded situations, we have developed another distributed scheduling technique called the CDP-BBA scheduling strategy. CDP-BBA explicitly accounts for both the deadlines and criticalness of tasks and exchanged messages when making scheduling decisions.

In order to evaluate these scheduling techniques, we have described our implementation and show experimental and simulation results. Moreover, our prototype implementation of these scheduling strategies in a middleware also prove their effectiveness.

In the next chapter, several aspects of the next generation of adaptive and feedback-based scheduling techniques will be presented. We present adaptive scheduling strategies to schedule dynamically driving assistance functions in presence of driving situation change.

# *Dynamic feedback scheduling for automotive environments*


## Abstract

The use of feedback and adaptive techniques has been gaining importance in the context of scheduling in real-time systems as a mean to provide predictable performance with respect to the dynamics of the environment in which the system is operating. Therefore, feedback scheduling can be used to adjust the resource allocation and track the system performance.

The objective of our work in this chapter is to develop and analyze feedback-based adaptive scheduling schemes for high-level vehicle applications. The proposed feedback-based scheduling schemes are devoted to schedule ADAS (Advanced Driving Assistance System) functions.

The adaptation in our scheme is carried out according to the *driving situation*, which will further lead to the change of the associated driving assistance function's *criticalness*. Thus the schedule will be adjusted, that is, we can obtain a schedule that may satisfy the desired real-time requirements.


## Contents

## 5.1 Introduction

The case study depicted in this chapter focuses on the *advanced applications for vehicles*. The in-vehicle applications may be classified in two main categories. The first category is the *vehicle control system* with high demands on safety, reliability, and accurate timing. The tasks in such systems have hard real-time requirements and are scheduled *off-line* and execute according to a dispatched table on-line.

The second category includes functions for diagnosis and infotainment that are used in vehicles to support maintenance, vehicle service, comfort, and driving assistance functions. Components in this category must be integrated without impacting safety critical functionality in the vehicle and may be scheduled *dynamically* at run-time rather than statically at design time. The application concerned in our case study falls in this category and it concerns more precisely the driving assistance functions.

The introduction of new technologies in and outside the car provides new opportunities to better support the driver when confronted to exceptional situations that may result in an accident. These new technologies can in particular prevent driver inattention or provide assistance on how to react.

The car industry is going now in a direction to define new ADAS (Advanced Driving Assistance System) to increase driver safety. In order to evaluate these ADAS functions, we need an estimation of driver behavior indicators, and robust models of Human Vehicle Interfaces (HVI). However, the evaluation and testing of new proposals before technology investments may constrain the HVI models (Figure 5.1).

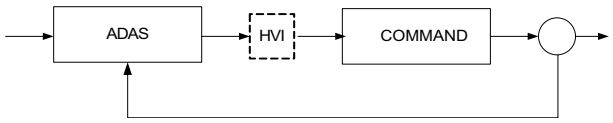



Figure 5.1: ADAS evaluation

As an example of existing ADAS, we consider the Adaptive Cruise Control (ACC). ACC is an extension of existing cruise control systems, designed to maintain the vehicle speed. The advanced features of ACC systems include the ability to track a car in the lane ahead using forward looking radar. If the distance to a vehicle in front is below a *safety distance* value, the ACC system is designed to slow the car down, to track the speed of the preceding vehicle, then to bring back the car to its pre-set speed once the lane ahead is clear. Steering angle and yaw rate sensors detect lanes and predict road curves, ensuring any vehicle in front is in the same lane as the target car.

One of the potential advantages of ACC is the foundation it provides for next generation advancements in lane detection systems, eventually expected to include cameras. The use of cameras in the vehicle is predicted to help provide for better lane following and collision avoidance by controlling the steering mechanism of the vehicle. ACC is now implemented on some vehicle models (e.g., BMW, Mercedes, Nissan).

An example of future ADAS that we can imagine is the Driver Hypovigilance (DH). DH intends to monitor the driver and the environment and will detect driver hypovigilance on line, based on multiple parameters. Using driver monitoring sensors (such as an eyelid movement and a steering grip sensor), the system monitors and evaluates the time $dt$ during which the driver does not look at the parebrise. If $dt$ is more than 10% of the last time unit $\Delta T$, then the system alarms the driver. $\Delta T$ may be function of several parameters such as the vehicle speed ($\Delta T \propto (1/\text{speed})$ ).

In the next generation of ADAS, the current driving situation and the driver behavior evaluation using high-level HVI metrics have to be integrated on the real-time ADAS control loop. On the other hand, this feedback assessment has imposed several requirements concerning for example the real-time

scheduling and feedback scheduling to adapt the computing resources to the current driving situation (Figure 5.2).

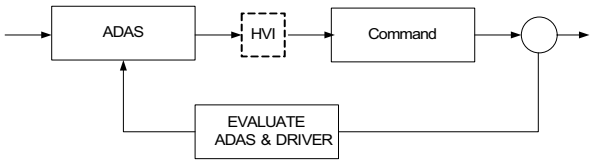



Figure 5.2: Adaptive ADAS

The static scheduling for future ADAS functions in automotive applications leads to over dimensioning the distributed computing resources. The dimensioning is based on the burst computing resources needed at the moments the ADAS should be activated, which are quite rare with respect to the normal operation. Moreover, in such systems, critical and less critical components coexist. The critical part of such systems is generally scheduled at pre-runtime by using static and cyclic techniques [Kop00][KG94]. The high-level part of the automotive system (e.g., ADAS functions) is less critical but requires dynamic reconfiguration and certain level of quality of service.

To deal with such complex systems, we propose to use the "Feedback" scheduling paradigm [JST99][PB00][Fer93] to adapt dynamically the available CPU and communication resources to the applications requirements. This technique is well adapted to perform in unpredictable dynamic systems, i.e., systems whose workloads cannot be accurately modeled. For example, a real-time system for an advanced vehicle is characterized by the unpredictable nature of the vehicle, road conditions and driving situations.

This chapter begins with an overview on the "Feedback scheduling" paradigm. Then we present our architecture and the general context of our automotive applications. Finally, we present our proposed methods and techniques to schedule dynamically ADAS functions based on the automotive applications *context*.

## 5.2 Feedback scheduling: state of the art

Many distributed real-time systems become more and more unpredictable due to several factors such as the increasing use of commercial off-the-shelf components, the unpredictable environment, etc. These systems interact with environments where both load and available resources are difficult to predict. Thus, feedback scheduling is introduced, to deal with CPU and communication resource variations and unpredictable workload during run-time.

This recent approach has been introduced by both the real-time computing field and the automatic control field. By combining scheduling theory and control theory, it is possible to achieve higher resource utilization and better control performance. To achieve the best results, co-designing, the scheduler and the controllers is necessary.

The idea is to introduce control tasks to the system, which are generally considered as hard real-time tasks with fixed sampling periods. The feedback scheduler may be viewed as a task that controls the processor utilization by assigning task periods that optimize the overall control performance. The controller task is viewed as a periodic task, with a period larger than the sampling periods of the system tasks. Thus, the controller task modifies the sampling period of tasks only when resource availability changes have been observed. Several research works have been conducted on the feedback scheduling, they fall into three categories.

### 5.2.1 Integrated control and real-time system design

In [SLSS96], sampling period selection for a set of control tasks is considered. The performance of a task is given as a function of its sampling frequency, and an optimization problem is solved to find the set of optimal task periods.

A conjuction of real-time scheduling theory and the control systems has been introduced in [RHS97], where the performance parameters are expressed as functions of the sampling periods and the input/output latencies. In [SM99] the authors deal with online rescaling and relocation of control tasks in a multiprocessor system.

### 5.2.2 Quality of service approaches in real-time systems

The second category of feedback scheduling concerns Quality of Service (QoS) aware real-time systems. In such approach, the system's resource allocation is adjusted on-line in order to maximize the performance with acceptable quality of service. In [JST99], the authors propose a framework that allows the control of application requests for system resources using the amount of allocated resources for feedback. It is shown that a PID (Proportional, Integral, Derivative) controller can be used to bound the resource usage in a stable and fair way.

In [AB99], Abeni and Butazzo proposed task models suitable for multimedia applications. Two of these models use PI (Proportional, Integral) control feedback to adjust the reserved fraction of CPU bandwidth. Several tasks are competing for finite resources, and each task is associated with a utility value, which is a function of the assigned resources. The system distributes the resources between the tasks to maximize the total utility of the system.

In [AAS97], the authors proposed a QoS renegotiation scheme that allows a graceful degradation in cases of overload, failures or violation of initial assumptions. Their solution allows the clients to express, in their service requests, a range of QoS levels they can accept from the provider, and the perceived utility of receiving service at each of these levels.

These adaptive scheduling algorithms are used for computing systems such multimedia [BGM+97], distributed visual tracking (to guarantee desired network packet rate) [LN99], operating systems [SGG+99] and communication systems [AS99].

### 5.2.3 Flexible and adaptive real-time system algorithms and architectures

The third category relates to the wealth of flexible scheduling algorithms available. An interesting alternative to linear task rescaling is given in [BLA98] where an elastic task model for periodic tasks is presented. The relative sensitivity of tasks to rescaling are expressed in terms of elasticity coefficients. Schedulability analysis of the system under EDF scheduling is given.

In [BGM+97], the authors evaluated a dynamic QoS manager by measuring the transient performance of applications in response to QoS adaptations. A set of metrics was proposed to capture the transient state performance and its impact on applications.

### 5.2.4 Feedback scheduling for autonomous vehicles

Related to our work, in [PB00] the authors propose a method of ranking the automotive services at pre run-time using the notion of "utility" of each service. They made the focus on the definition of

utility values of services, their computing and then their assignment to real automotive applications. Their approach lacks of real implementation and effective distributed scheduling.

In [LMS04], the authors propose a feedback-based adaptive scheduling schemes for autonomous vehicle systems. They consider a system with mobile nodes. When the nodes are mobile, they continuously move and execute certain tasks, so the mobility can affect the task parameters. Thus, they identify the relation between the mobility characteristics (e.g., speed) and the values of task parameters (e.g., execution time, deadlines and periods) for a particular application. The node in their system is a vehicle which is equipped with sensing, processing, and actuating capabilities. Adaptation is focused on the speed of the vehicle, which will lead to changes of task parameters (deadlines and periods of tasks).

Moreover, computing resources used inside an autonomous vehicle are limited, and may fail. Thus, we must adapt the available resources according to the criticalness of services and to select the services dynamically. A dynamic and adaptive scheduling strategy may lead to a better utilization of resources and to select the important services regarding the system configuration.

To evaluate a feedback scheduling technique, it is necessary to have simulators that allow joint simulation of continuous time plant dynamics, discrete time controllers, and the real-time scheduling of the corresponding controller tasks. The simulations of our feedback scheduling techniques are based on the Matlab/Simulink toolbox "truetime" presented in [HCr03].

The objective of our work in this chapter is to develop and analyze feedback-based adaptive scheduling schemes for autonomous vehicle systems. In particular, we are interested in the scheduling of the driving assistance functions in case of driving situation change. Thus, we have to identify the relation between the driving situation (e.g., overtaking or fluid driving on motorways) and the parameters associated to driving assistance functions (e.g., criticalness or utility value).

The adaptation in our scheme is carried out on the *driving situation*, which will lead to the change of the associated driving assistance function's *criticalness*. Thus the task scheduling will be adjusted to satisfy the desired real-time requirements.

The software architecture of our system considered here consists of several computers networked by a communication bus, which communicate between them using a middleware.

## 5.3 Our architecture for advanced autonomous vehicles

In an advanced vehicle control system, *safety and critical* services are scheduled statically using an execution order established *offline*. For the case study presented in this chapter, we consider the less critical services such as ADAS functions. This class of functions processes a large amount of data and requires a high-level of reconfiguration.

In order to assess and evaluate at run-time these ADAS functions, several sensors are embedded in our demonstrator car STRADA in order to elaborate metrics to evaluate these ADAS. Our framework SCOOT-R is used as the software architecture to acquire, process and display these metrics. We consider here only the client/server model of SCOOT-R.

In order to schedule at run-time such ADAS functions and metrics, we have adapted the feedback scheduling technique to implement adaptive and distributed scheduling strategies. The idea is to carry out the adaptation on the *driving situation* and then to re-schedule computing of the associated metrics based on their criticalness.

In the framework of the RoadSense project, several metrics were defined and implemented. These metrics relate to the safety, comfort, and support assessment. The performances of the driver with

respect to the assistance system are thus evaluated using these metrics.

Hereafter, we describe the *driving situation* and associated *metrics* used in this case study.

### 5.3.1 Driving situations and metrics definition

In order to provide a relevant assistance to the driver, it would be necessary for it to be individualized, and adapted as much as possible to the driving situation. This implies that the on-board computer should be able to recognize the driver behavior and the driving situation. The assistance can be also adapted to the model of the vehicle, and can evolve with its life. To take all these parameters into account, the real-time computation of the metrics seems to us a relevant prospect.

The driver behavior is very dependent on the driving situation, including:

- the dynamic state of its vehicle (speed, acceleration);
- the state of its vehicle according to the static environment (position on the road, position on the lane);
- the type of the static environment (highway, number of lanes, type of lines, etc.);
- the state of the dynamic environment, that means the situation (position and speed) of the other road users.

Moreover, each driving situation evokes a set of metrics. The relative criticalness of these metrics change from a driving situation to another.

The metrics are grouped in several categories:

- side and longitudinal controls by the driver;
- visual management, gathering the parameters relating to the direction of the driver glance;
- interactions with other vehicles;
- conscience of the driving situation (overtaking, fluid driving on motorways, etc.);
- the direct reaction of the driver with respect to the assistance system.

In order to illustrate the type of calculations necessary to obtain these metrics, and to better represent the principle of the temporal relations between them, we expose below some of these metrics:

**Time Headway (TH):** It is the time it would take the host vehicle to arrive in place of the leading vehicle without changing the speed. This measure is computed as the bumper-to-bumper distance divided by the speed of the host vehicle. This metric is coded at run-time.

**Mean Speed (MS):** This is the vehicle's average speed over a given time period. For Renault case study, the mean speed was in km/h. This metric is coded at run-time for a time period equal to 0,5 seconds.

**Time To Collision (TTC):** The time to collision is the delay before the host vehicle reaches the preceding one if none of them changes its speed. It can be computed by the above formula:

$$Time\ to\ collision(s) = Following\ Distance(m)/Relative\ Speed(m/s) \tag{5.1}$$

In this case study, we identify a set of driving situations S = $\{S1, S2, S3, ...\}$, determined statically at pre run-time by human factors experts. A driving situation is composed of a set of metrics M = $\{M1, M2, M3, ....\}$. Each metric is a useful information that indicates a relevant measure about the vehicle and the driver behavior. A metric can be computed by one or more data sensors or the fusion of many information sources.

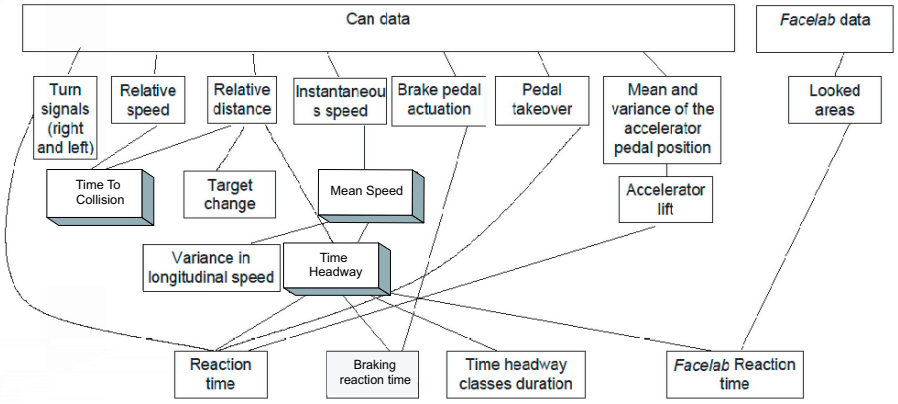



Figure 5.3: A real example of metrics architecture

An important type of constraints for real-time tasks is the precedence one. A possible technique of scheduling with precedence constraints would be to assign the criticalness to the tasks according to the constraints of precedence.

This technique seems interesting in our study case regarding the hierarchy of computing high-level metrics (Figure 5.3). The programmer has to give in this case the precedence relations between metrics and then the system computes the relative criticalness values.

As shown in Figure 5.3, for example to compute the "Mean Speed" metric we need to have the "Instantaneous Speed" one. Moreover, to compute the "Time Headway" metric, we need the "Mean Speed" value. Thus, the "Instantaneous Speed" component must have a criticalness greater than the "Mean Speed" and "Time Headway" components.

Moreover, several computation methods of one metric could exist, either based on the same data sensors, or on different sensors.

Switching from a driving situation to another is monitored by an independent module (CASSICE module). The CASSICE (CAractérisation Symbolique de SItuations de ConduitE) [YL05] module was developed in the framework of ARCOS action (Prédit national (2001-2004)). The driving situation, like overtaking maneuver, regular driving, etc., is recognized using states sequence techniques coupled to the belief theory.

This module allows the detection of driving situation change and then broadcasts this information to all the nodes of the system. Thus, all nodes are informed by this change and apply the assignment of the criticalness values to the related components.

### 5.3.2 Distributed computing architecture

The computing architecture in this case study is distributed on several computers. The client/server model of SCOOT-R is used to exchange data. Table 5.1 shows how the components of the system are located and distributed on computers.

| Components / Computer | Clients | Servers |
|---|---|---|
| Computer1 | No-Clients | Instantaneous Speed (IS) server, Relative Speed (RS) server, Relative Distance (RD) server |
| Computer2 | Relative Speed client (Client 1) , Relative Distance client (Client 2) | Time To Collision (TTC) server |
| Computer3 | Instantaneous Speed (IS) client (Client 3) | Mean Speed (MS) server |
| Computer4 | Mean Speed (MS) Client (Client 4), Relative Distance (RD) client (Client 5) | Time Headway (TH) server |

Table 5.1: Computers and their associated components

In Figure 5.4, two metrics are illustrated: TTC (Time To Collision) and TH (Time Headway). In order to compute a metric, we have to identify the components that contribute to elaborate it. For example, to compute the TTC metric, we identify two components, component 1 and component 2. Each component contains several servers and clients and others tasks. Component 1 contains two servers that provide CAN data "RS" and "RD" servers. Component 2 contains two clients (client1 and client2) for the two servers "Relative Speed (RS)" and "Relative Distance (RD)" in component 1. It also contains a task that computes the Time To Collision (TTC) and finally a server "TTC" that provides this metric to other components.

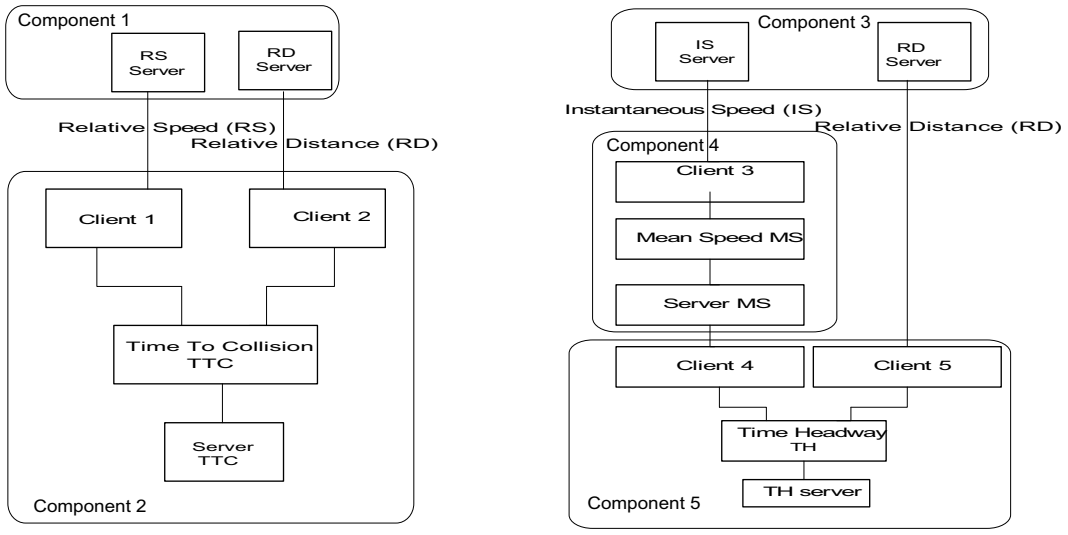



Figure 5.4: Computation architecture of metrics

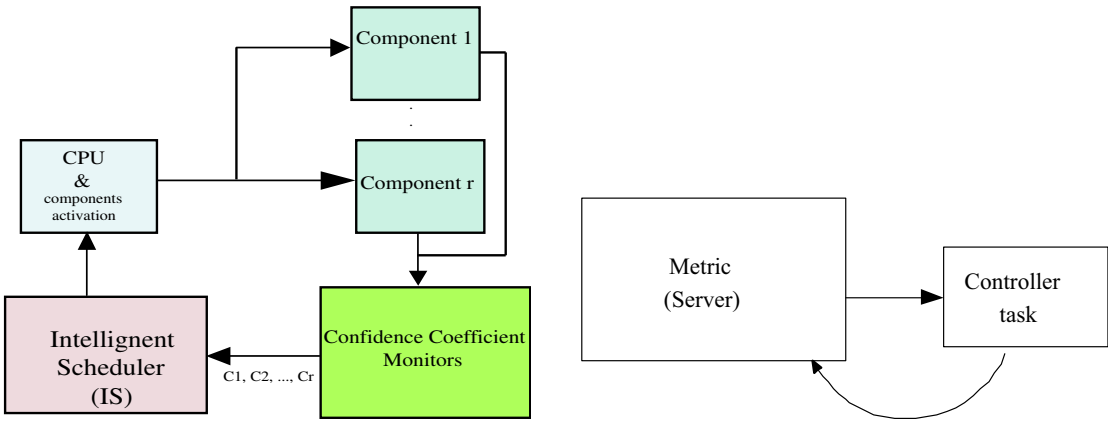
Thus, one component may have none or several clients, depending on its data requirement. One component has always a server to provide the data it produces.

## 5.4 Feedback scheduling of tasks and messages

While the driving situation changes dynamically, the criticalness of the associated metrics changes too. For example, overtaking on a two-way road and on motorways are two driving situations that have different consequences on the high-level metrics criticalness. So, affecting static criticalness to components is not suited for applications where components importance depends on the contextual situation. Thus, we propose to use feedback scheduling paradigm to adapt the criticalness of components based on the application context.

In order to design and implement feedback scheduling techniques, two schemes are proposed. First, the "Upward scheme", where we evaluate the low-level data *confidence coefficient* and then map it to the high-level behavior of the associated service. In this scheme, SCOOT-R quality indicator is used to select the active service and to evaluate a concrete *confidence coefficient* of the service.

Second, we propose the "Downward scheme". In this case, we identify a set of driving situations assumed to be known *a priori*. Each driving situation calls up several metrics. During a specified driving situation and given the set of metrics used in this situation, we can compute at pre run-time the criticalness of all the metrics by comparing their relative importance. Now that the services criticalities are established, we use the distributed scheduling strategy proposed in the previous chapter (CDP-BBA) to apply the scheduling policy in a distributed environment. Moreover, we can use the criticalness profile diffused by the CDP-BBA algorithm to compute dynamically the criticalness of part of the components. The architecture of a feedback scheduler is shown in Figure 5.5(a).




(a) Feedback scheduler structure (b) Confidence coefficient assessment

Figure 5.5: Adaptive Scheduler

In order to express the criticalness of a metric, we use the notion of *confidence coefficient* that is computed dynamically and described below.

### 5.4.1 Confidence coefficient of metrics

In order to deal with dynamic systems, we introduce the notion of *confidence coefficient* that reflects data quality of the associated component.

We associate the *confidence coefficient* to each metric. This coefficient is obtained by the fusion team of our laboratory in order to take into account the data imprecision of sensors [Bez05]. Let's

recall that a metric may be computed by various techniques; we compute the *confidence coefficient* that corresponds to each technique.

In general, this coefficient may depend on several criteria, such as the quality of data (e.g., accuracy, precision), the environment status (e.g., visibility, going beyond on a motorway), etc.

In order to illustrate the relevance of our concept, we present the vehicle speed computation by two different sensors. By computing the distance between two GPS acquisitions (at 1 Hz), the degree of confidence may be obtained from the uncertainty ellipse of the position delivered by the GPS. The second technique uses the odometers on the wheels; we compute the mean value of n precedent values of the speed from the odometer:

$$V = \sum_{i=1}^{n} \frac{Vi}{N} \tag{5.2}$$

We thus obtain the speed for each wheel. The vehicle speed is the average of these four speeds. In order to obtain the degree of confidence, we considered it inversely proportional to the difference $\Delta V$ between the highest $Vh$ and smallest $Vs$ speed among the four wheels. This estimation seems correct as the speed variation is proportional to the loss of adherence. We thus obtain:

$$\Delta V = Vh - Vs; \qquad and\ confidence\ coefficient\ = \frac{1}{\Delta V} = k * \frac{1}{\Delta V} \tag{5.3}$$

The idea of using the *confidence coefficient* for achieving flexible behavior has been promoted in the real-time literature. It is called *utility value* in some related research works [PB00]. Hereafter, we present how we compute the *confidence coefficient* in a given driving situation. Then, we describe the dynamic assignment and scheduling of these metrics and their components.

### 5.4.2 Upward scheme: feedback scheduling using SCOOT-R quality indicator

In this scheme, the adaptive scheduling of metrics concerns particularly the dynamic assignment of *confidence coefficient* to the metrics. There is a controller task associated to each service of the system, which monitors and evaluates the *confidence coefficient* of the service at low frequency (Figure 5.5(b)).

In order to obtain a concrete *confidence coefficient* value of a metric, we have adapted the notion of *quality indicator* of SCOOT-R. Let's recall that a SCOOT-R quality indicator is useful in case of multiple available servers for the same service. This indicator permits to select the server having the best quality. Other servers for the same service switch to a mode of "standby". They decrease their consumption and requirements of CPU resources (frequencies and deadlines) by applying a predefined rule.

To illustrate this method, let's suppose that we have the "vehicle speed" metric provided by two different services (SCOOT-R servers) T1 and T2. T1 computes the vehicle speed using GPS and T2 service by using the vehicle odometers. When the *confidence coefficient* varies (resulting from GPS mask/unmask, connection/disconnection of an odometer of the wheels, etc.), the service switches between two modes, "Active" and "Standby" modes. We notice that the decrease of the *confidence coefficient* of the service T1 leads consequently to the decrease of its activation frequency (Standby mode). This decrease of the *confidence coefficient* in T1 service is compensated by the increase of the T2 one and consequently the increase of its activation frequency (Active mode) (Figure 5.6).

Since our architecture is based on a SCOOT-R client/server model, a metric in our architecture contains a server that will provide the service. Thus, the *confidence coefficient* of a metric is mapped to its server task (SCOOT-R quality indicator).

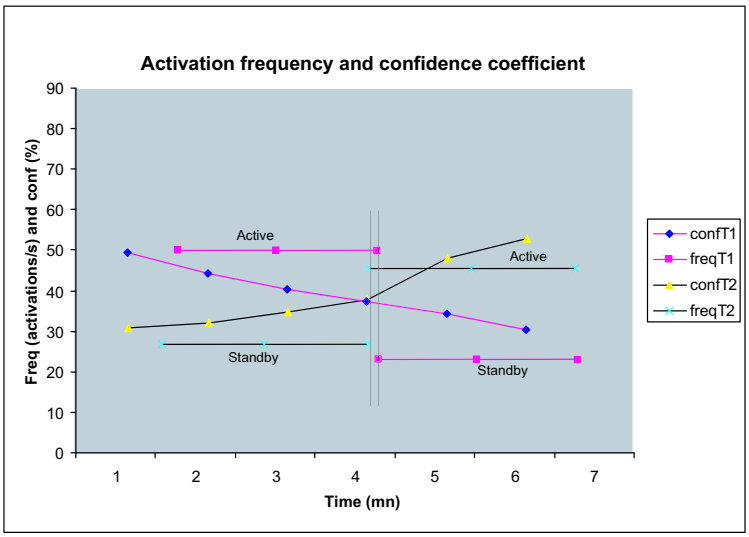



Figure 5.6: Activation frequency and confidence coefficient

The major disadvantage of this scheme is the need to compute the metric by several techniques in order to update the value of the *confidence coefficient* associated to each one. In order to limit this computing overcost, the data fusion team could obtain the *confidence coefficient* of metrics independently of the metric value.

### 5.4.3 Downward scheme: feedback scheduling regarding driving situation

In this scheme, we consider the run-time assignment of criticalness to the metrics of the current driving situation. This assignment is based on the change of driving situation. Many criteria may affect the driving situation, for example the state of the environment (e.g., 'wet' or 'dry' road conditions; 'day' or 'night' visibility). The switch from a driving situation to another will modify the set of metrics used within the driving situation and the criticalness of these metrics.

As mentioned before, an independent module allows the detection of the driving situation change (CASSISE module). It informs all the nodes by broadcasting this change. Each node of the system contains a table that stores the criticalness of each local metric within a specified driving situation. Thus, in case of change in the driving situation, each local node applies its policy using the local table and assigns the criticalness to metrics based on the driving situation. The scheduling techniques follow one of the two strategies described below.

#### 5.4.3.1 Using a finite number of criticalness classes

In this scheme, we consider a finite number of criticalness classes. Thus, we assign to each metric certain level of criticalness (Figure 5.7). Let's recall that during a driving situation, the criticalness of each metric remains constant.

In Figure 5.7, we consider for example three set of ADAS functions (driving situation, vehicle diagnostic functions, and alarm functions), which are mapped to three level of criticalness.

However, we have defined four classes of criticalness to cover the most automotive high-level applications requirements. These classes are:

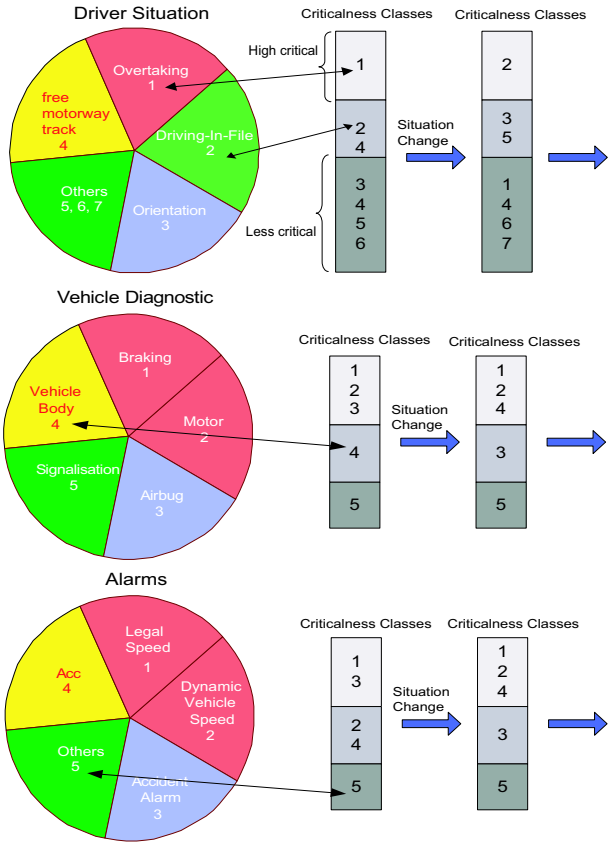



Figure 5.7: Driving situation switch

- Highly Critical (class 1): highly critical metrics to be computed;
- Critical (class 2): must be guaranteed excepted in the case of system failure (hardware reconfiguration, etc.);
- Less Critical (class 3);
- non-critical (class 4).

Class 1 includes the most critical functions of the system such as *decisive metrics* for the current driving situation. The services inside this class must satisfy their requirements whatever the situation is (excepted in case of hardware failures).

In class 2, the services are considered critical and must satisfy their timing requirements in case of normal operation of the system. In case of partial failure or another incidents in the system, this part of services may miss some deadlines. Thus, these services are qualified as strict real-time in absence of incident and soft real-time in presence of incidents. These services include the most metrics for the current driving situation.

The class 3 concerns the soft real-time services of the system. The services of this class are not hard real-time but require some level of correctness. This services concerns generally the metrics of no interest with the current driving situation.

In class 4, the services are not real-time (e.g., telematic functions). The services of this class are considered to be sacrificed in case of overload and failure situations.

#### 5.4.3.2 Using a continuous scale of criticalness

In this case, we consider a continuous scale of criticalness instead of finite number of classes. The main advantage of this approach is that there is no restrictions on the criticalness levels of components, i.e., the programmers may express their specifications without restrictions. We will use the notion of "utility value" to define the criticalness of a metric.

The main objective in this scheme is to obtain a utility value for each metric that can be used for best benefit scheduling. Given the set of metrics for a particular driving situation, we can generate an ordinal measurement of values using pair-wise comparisons and preference judgments, through a sort of dichotomy.

To compute the utility value, binary relations, commonly found in set theory can be used to compare the utilities of different metrics. Let "More critical" be such a relation:

$$S1 \ More \ critical \ \ than \ S2 \Rightarrow V(S1) \geq V(S2) \tag{5.4}$$

Where V is the "value function" that assigns a real number to a driving situation. If V is defined for only a part of S, then it is a partially ordered utility value function. If V is defined for all Si in S, then it is called an ordinal value function.

Hereafter, we present an illustrative example of two driving situations "$National - Road - 2 * 1$" (two-way road) and "$Driving - In - File$". Each driving situation contains several metrics that provide the relevant information (Figure 5.8).

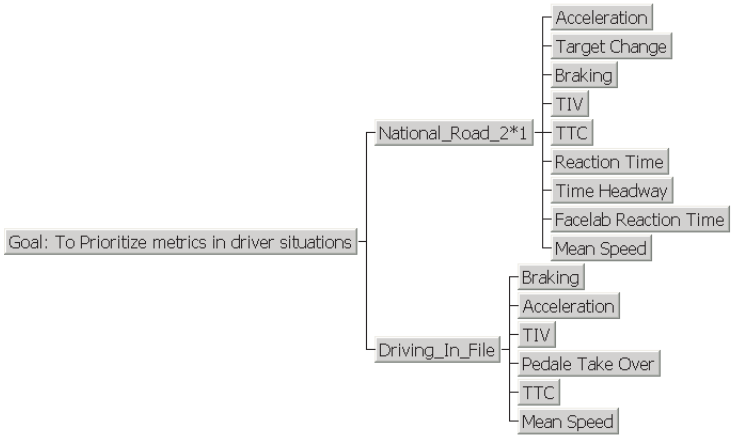



Figure 5.8: Driving situations "$National - Road - 2 * 1$" and "$Driving - In - File$" and their associated metrics

In order to obtain an ordinal value of utility for each metric, we proceed by pairwise comparison based on preference judgments between metrics. The domain expert decision maker can be asked to express the preference relation between every pair of metrics available in a specified driving situation. If the expressed preferences are consistent, then an ordinal value function can be constructed to represent them.

Let's note that the values are computed locally and globally. We consider that during a driving situation, the utility values of metrics are computed and remain constant in the same situation. These values are computed by two ways: (1) values are computed locally using the metrics for the current situation, (2) values are computed globally using all metrics of all driving situation that may exist in the system.

We used Expert Choice (EC) software to perform the pair-wise comparisons and built up an internal representation of preference judgments. EC is a independent shareware running under Windows that supports various graphical and user-interface methods to enable these pair-wise comparisons to be entered [EC886]. The comparison procedure is easy to carry out provided that there is a field knowledge and engineering data available to support the comparisons.

Figure 5.9 illustrates an example of a driving situation "$National - Road - 2 * 1$". The utility value of metrics are obtained locally using the pairwise comparison within the "$National - Road - 2 * 1$" driving situation.

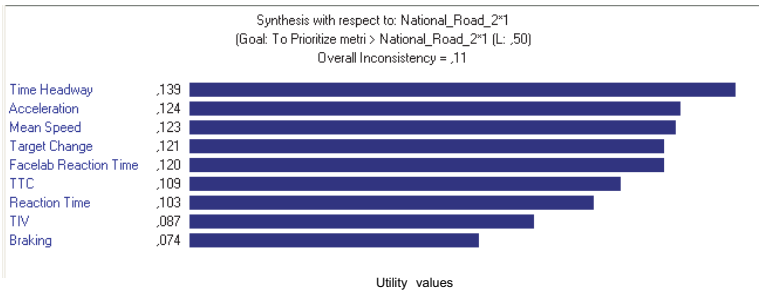



Figure 5.9: Utility values for "$National - Road - 2 * 1$" driving situation obtained by pairwise comparison

In Figure 5.10, an ordinal value of metrics utility is obtained for all the driving situations. The pairwise comparison is made in two steps. First, by comparing the driving situations, and then, the comparison of the metrics in each driving situation.


Figure 5.10: Utility values for all metrics of all driving situations obtained by pairwise comparison

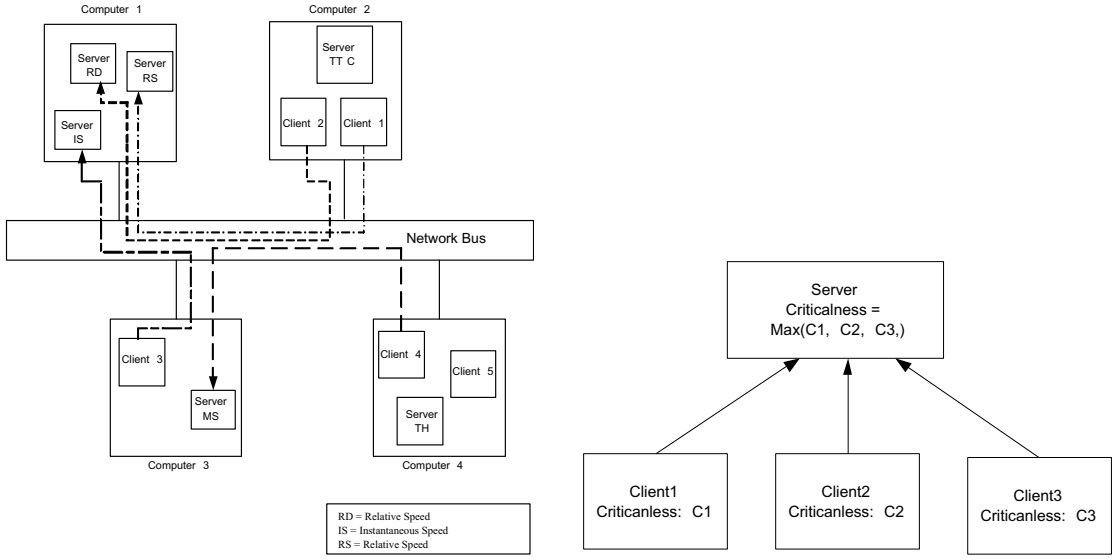
The utility values of all metrics are obtained for a specified driving situation. Hence, we can apply the CDP-BBA algorithm to schedule dynamically the transactions and associated tasks.

Choosing the utility values of all metrics at run-time regarding a specified driving situation is prohibitively time consuming. To overcome this, we propose to compute statically and off-line the utility values.

The set of driving situations is supposed to be known *a priori* and the associated metrics too. Thus, at pre run-time, the field expert performs pairwise decisions and then obtains the desired values. These values remain constant for a given configuration.

#### 5.4.3.3 Using CDP-BBA scheduling strategy to schedule the metrics and associated components

Now that the assignment of criticalness to metrics has been established, using a *finite number* of criticalness classes or using *continuous scale* of criticalness, we can then apply the CDP-BBA distributed scheduling strategy to schedule these metrics and associated components. As our distributed computing architecture is based on client/server model, the scheduling of transactions using CDP-BBA is coherent with our case study presented here. Figure 5.11(a) shows the computing architecture using a client/server model.




(a) Distributed computing architecture (b) Server criticalness inheritance from its clients

Figure 5.11: Computing architecture and criticalness inheritance

The feedback scheduling may be implemented directly using CDP-BBA. It will consider the criticalness and the deadlines of messages while making a scheduling decision. The use of a client/server model in our architecture has lead to precedence relations. For example, client4 in computer4 must wait the response from the server "MS" on computer3. The server "TTC" on computer2 must wait also the responses from the "RS" (Relative Speed) and "RD" (Relative Distance) servers (computer1) to compute a new value of "MS" (Mean Speed). This precedence scheme introduces constraints on the criticalness of the entities of the system. Let's give the case of a client and its server, we suppose also that the client and the server belong to two different components. It is the case of the client4 and the "MS" server in our architecture. The client waits for the server response. Initially the client and the server have different level of criticalness since they belong to two different components. Let's suppose that the client has a criticalness greater than the server one. This leads to a logical contradiction. To solve this problem, the simple way is to assign to the server the highest criticalness of its clients (Figure 5.11(b)).

The server examines the client request and extracts the criticalness parameter from the message header. Then, it compares it to its current criticalness and sets the highest value.

As shown in Figure 5.12, the distribution of decision is totally transparent and the scheduling decisions are made on each node of our computing architecture using the same rules.

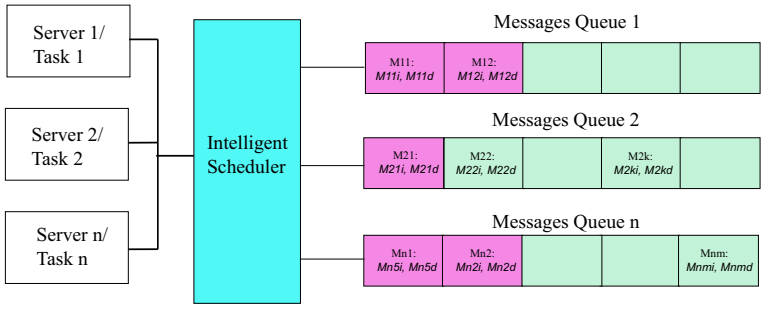



Figure 5.12: Using BBA to distribute adaptive scheduling

## 5.5 Simulation results

In order to illustrate the performance of the proposed feedback scheduling techniques, we have modeled the computing architecture presented in section 5.3.2 using the TrueTime toolbox of Matlab.

The architecture comprises four computers. We have implemented the client/server model of SCOOT-R in TrueTime Matlab toolbox to exchange data between the components. Table 5.2 shows how the components are located and distributed on computers.

| Components / Computer | Clients | Servers |
|---|---|---|
| Computer1 | No-Clients | Instantaneous Speed (IS) server, Relative Speed (RS) server, Relative Distance (RD) server |
| Computer2 | Relative Speed client (Client 1) , Relative Distance client (Client 2) | TTC server |
| Computer3 | Instantaneous Speed client (Client 3) | Mean Speed (MS) server |
| Computer4 | Mean Speed (MS) Client (Client 4), Relative Distance client (Client 5) | Time Headway (TH) server |

Table 5.2: Computing architecture

The computing architecture is thus composed of several components, each component contains several clients and servers and processing tasks. Figure 5.13 illustrates the communication between these components.

To detect driving situation change, a fifth computer (monitor computer) is added. Monitor computer contains a periodic task that broadcasts a message each 10 seconds to announce a driving situation change.

Moreover, each computer contains an independent aperiodic task called "status change". This task detects the reception of a driving situation switch message and then extracts the status information from it to map the associated criticalness to the local component using a local static table (Figure 5.14).

To assign the criticalness of components on a local node, each computer contains a static table that contains all the driving situations and associated components criticalness (Tables 5.3 and 5.4). The criticalness values are computed at pre run-time.

To execute the components based on their criticalness and deadlines, we used the CDP-BBA algorithm. We conducted simulation studies to determine the effect of driving situation change on

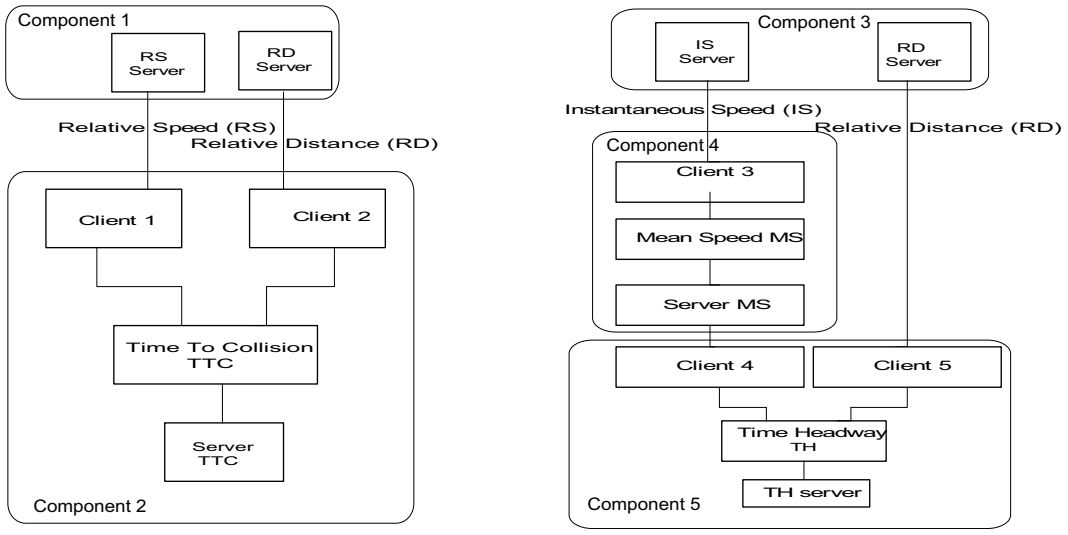



Figure 5.13: Components interactions and metrics computation architecture

| Driving situation \ Components | Client RS (Client1) | Client RD (Client2) | TTC server |
|---|---|---|---|
| Driving Situation S1 | 1 | 1 | 2 |
| Driving Situation S2 | 2 | 3 | 3 |
| Driving Situation S3 | 1 | 1 | 2 |

Table 5.3: Driving situation and associated criticalness on computer 2

| Driving situation \ Components | Client RS (Client1) | Client RD (Client2) | TTC server |
|---|---|---|---|
| Driving Situation S1 | 3 | 3 | 4 |
| Driving Situation S2 | 2 | 1 | 3 |
| Driving Situation S3 | 3 | 3 | 3 |

Table 5.4: Driving situation and associated criticalness on computer 4

the components performances; the criteria used to measure the performance is the *response time* of transactions.

Figure 5.15(a) shows response times in seconds of client2 in case of multiple driving situations. While respecting the criticalness values depicted in Table 5.3, we show that client2 has a criticalness value of 1 in the driving situation S1, whereas client5 has a criticalness 3 in this situation (Table 5.4). The component having the lowest value of criticalness is the most critical. Thus, as shown in Figure 5.15(a) and Figure 5.15(b), the average response time of client 2 is smaller than client 5 during the S1 driving situation, whereas during S2 driving situation, client 5 has a response time average smaller than client 2.

Let's note that in the absence of a feedback scheduling technique, those response times would be equal to the highest value 0.015 seconds instead of 0.01 seconds.

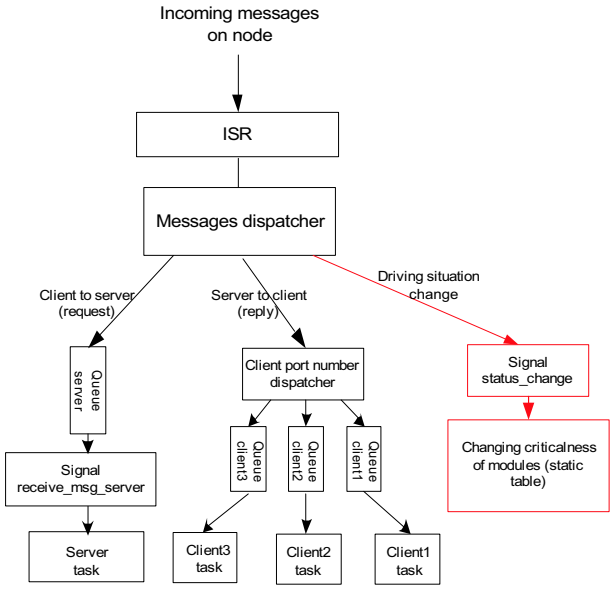
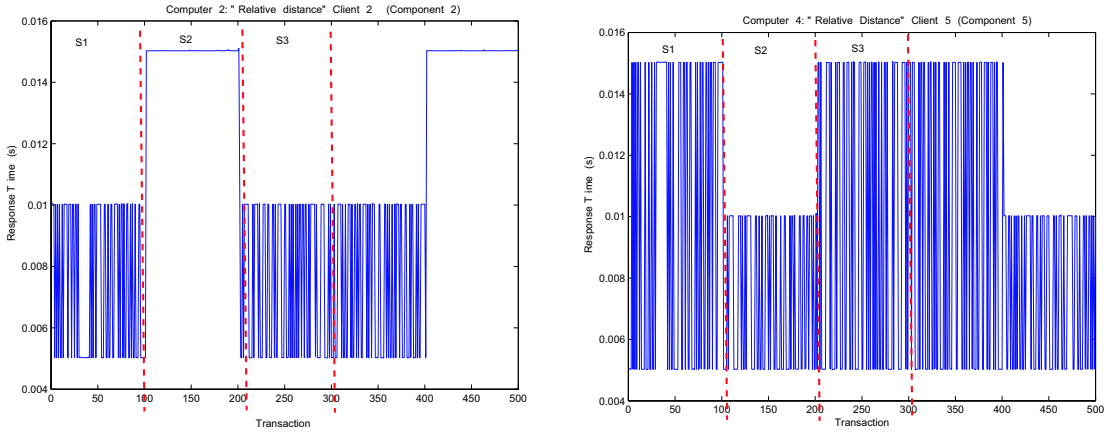



Figure 5.14: Dispatching incoming messages to clients and servers




(a) Response time of client 2 regarding driving situation change

(b) Response time of client 5 regarding driving situation change

Figure 5.15: Transactions response time

## 5.6 Conclusion

For the case study depicted in this chapter, we have identified a set of driving situations, determined statically at pre run-time. Each driving situation is related to a set of metrics. A metric is a useful information that indicates a relevant measure about the vehicle and the driver behavior.

We have developed and analyzed feedback-based adaptive scheduling strategies to re-schedule the metrics in case of driving situation change. Thus, we identified the relation between the driving situation (e.g., overtaking or fluid driving on motorways) and the parameters of the associated metrics components (e.g., criticalness or utility value).

The adaptation in our scheme is thus carried out on the *driving situation*, which will further lead to the *criticalness* change of the associated metrics components. Thus the schedule will be adjusted so

that we obtained a task scheduling that satisfies the desired real-time requirements.

When the criticalness values of the metrics components inside a specified driving situation are computed, we can apply our scheduling technique CDP-BBA to re-schedule the metrics components in a distributed environment while taking into account their criticalness values.

The driving situation change is detected by a separate module that broadcasts this change information to all the nodes of the network. Then, the *supervisor* module on each node examines this information and then applies the associated scheduling policy to its local components.

The simulation results show an amelioration of the worst response times for clients with higher criticality with respect to the current driving situation.

We are also able to design these feedback-based schemes by applying other scheduling techniques such as MUF [SK91a], DASA and LBESA [LRWK04].

On the other hand, the run-time detection and diffusion of the driving situation change is prohibitively time consuming.

# *Conclusions and Perspectives*

## General context

The research works depicted in this thesis have been conducted at *Heudiasyc* laboratory, among the *advanced vehicle team*, with the support of the European project RoadSense whose main objective was the development of an evaluation framework for new Human Vehicle Interfaces (HVI) strategies. A certain number of metrics was defined describing the reaction of the driver with respect to the assistance system, the performances of the driver are thus evaluated, with and without the assistance system, in order to assess its true benefit on the improvement of safety. The objective is thus to provide the data-processing tools necessary to the calculation of these metrics.

Part of our laboratory contribution to the project was the design of a distributed real-time system called D-BITE (Driver Behavior Interface Test Equipment). D-BITE permits to compute driver behavioral metrics for safety, comfort and support assessment. D-BITE system uses our SCOOT-R middleware to ensure the real-time communication between the perception sensors, fusion elements, decision modules, and the man-vehicle interface.

## Communication sub-system and middleware services: SCOOT-R

The large amount of data to be acquired and processed, the distributed environment of the applications, and the need of dynamic reconfiguration and synchronisation mechanisms, all these requirements have lead us to the development of the middleware SCOOT-R.

SCOOT-R enables application programmers to design and develop distributed real-time applications. It is layered between the application and the OS kernel and it handles, on behalf of the application, the temporal correctness, real-time communication and synchronisation in a distributed environment. SCOOT-R supports two different types of communication: *request/reply* interaction (client/server model) for asynchronous applications as well as real-time *streaming data* (emitter/receiver model) for isochronous applications. While SCOOT-R client/server model was conceived to respond to asynchronous applications, the emitter/receiver model is introduced to cover synchronous applications with high bandwidth, such as image acquisition, high rate analog signals acquisition, etc.

SCOOT-R ensures also a dynamic reconfiguration by replicating software components. The redundancy management in SCOOT-R does not require specific efforts of design and development to dynamically replace a server or emitter or to activate a redundant function. This enables an evolution of the services without interruption.

Using the IEEE-1394 bus as communication media and the RTAI kernel as operating system, the worst case client/server transaction delay for remote communication (client and server located on separate computers) is less than 100 $\mu$s without other activity on the IEEE-1394 bus. Furthermore, The recovery time when a server or emitter is removed and replaced by another (with higher quality) is in worst case 120 $\mu$s without other communication activity or servers/emitters registering activity.

The contribution of the thesis to SCOOT-R was related to the evolution of the temporal SCOOT-R model (temporal contract and rules). We have contributed to the development of the *emitter/receiver model* that was added to the basic client/server model to provide a complete distributed solution with the support of data flow applications (e.g., real-time image acquisition and processing, analog signal acquisition, etc.). Finally, the thesis has contributed to the *development methodology* of user applications and the *worst case analysis* of the distributed system.

SCOOT-R was used for the RoadSense project to acquire embedded sensors data in run-time and to perform post-processing in order to elaborate the necessary metrics and indicators. The SCOOT-R use is justified by the need of distributed, modular and flexible architecture that provide checking and synchronisation services.

Using SCOOT-R for this type of applications has shown us the need to provide a certain guarantee of the proposed communication services. For this reason, a new research topic concerning the system dependability is currently initiated. There is a double goal to this work: First, to prove that SCOOT-R satisfies the constraints on its reliability. This means to verify that no message can be lost, delayed or altered in case of occurrence of any internal or external event. Second, in a more general way to show that a system integrating this middleware layer can be considered as reliable and thus providing the required services in the defined time interval. This is why a formal SCOOT-R model is under construction with a colored temporised petri network.

Moreover, we are currently working on the development of an embedded version of SCOOT-R using OSEK RTOS on PowerPc (MPC555).

## Distributed scheduling

The initial version of SCOOT-R does not provide any mechanism for clients to indicate the relative scheduling parameters of their requests to SCOOT-R endsystems. This feature is necessary, however, to minimize end-to-end priority inversion, as well as to bound latency and jitter for applications with real-time QoS requirements.

Therefore, we have presented in chapter 4 our contribution to develop distributed scheduling strategies defining the end-to-end priority and timeliness propagation of distributed transactions.

Two distributed scheduling strategies are proposed, the Client Priority Propagation (CPP) and the Client Deadline Propagation (CDP). Then, we have compared these scheduling strategies with the trivial FIFO scheme. A mechanism of priority inheritance is implemented to avoid the priority inversion, i.e. the priority of the client task is mapped on the associated server task.

CPP provides, like the RM scheduling strategy, a schedulability *assurance* prior to run-time for invocations with higher priorities in case of overloaded situations. On the other hand, CPP offers an equivalent performance for lower priorities invocations.

When CPP is useful for transactions in fixed-priority applications, CDP reflects perfectly the timing profile of distributed transactions. Clients in CDP may express their timing profile by an arbitrary *absolute* deadline and thus a coherent scheduling may be implemented on all the nodes. Like CPP, CDP guarantees a high schedulability assurance for invocations with higher priorities (earliest deadlines).

Moreover, from a methodology point of view, CDP is well adapted to our middleware and more widely to any component-oriented approach (provided that we have an accurate global time). CDP is also relevant in case of mixed networks when several nodes apply the CDP policy and other limited nodes work with the FIFO scheme.

Unfortunately, the weakness of the CDP model is the performance degradation in case of overloaded

situations. Like all the purely dynamic scheduling algorithms (e.g., EDF, MLF), a transient overload in the system may cause a critical task to fail, which is not desirable for a dynamically reconfigurable system.

Thus, to overcome this limitation and to deal with overload conditions, we have developed a *hybrid* static/dynamic scheduling strategy, the Best Benefit (CDP-BBA) strategy. CDP-BBA allows the integrated tasks/messages scheduling while taking into account the deadlines and criticalness of tasks and associated messages. CDP-BBA guarantees the temporal requirements of the mandatory tasks and associated messages. It has to respect the deadlines and to consider the criticalness of the tasks in case of overload.

The proposed scheduling techniques are well adapted for transactional communication model (e.g. client/server) and are implemented using our SCOOT-R middleware. Our prototype implementation of these scheduling strategies in a middleware shows their effectiveness.

The design and implementation phases of new scheduling strategies and their integration in a distributed environment were significally reduced, proving the relevance of SCOOT-R middleware technology.

Our experimental results show that the original SCOOT-R implementation, in case of high CPU load or many concurrent clients on the network, cannot preserve end-to-end priority and deadline, and thus leads to high latency bounds.

Using the scheduling strategies developed in this thesis, the priority and deadline are propagated from end-to-end, thus avoiding the priority inversion phenomena. Furthermore, the client having the closest deadline misses fewer deadlines in case of increasing workload.

As future perspective in this direction, we plan to establish a complete framework that allows the development and integration of new scheduling techniques in a distributed environment. The main goal is to provide the programmers by a complete set of tools and services that will permit to design and develop real-time scheduling algorithms for real applications. In the mean time, we plan to integrate the (m-k) firm scheduling algorithm to take into account the criticalness of tasks/messages. Thus, our objective is the fast prototyping and implementation of real-time scheduling techniques in a distributed environment using the middleware approach.

## Feedback scheduling

Real-time systems for advanced vehicle are usually characterized by the highly dynamic and non-deterministic environment. The inherent non-determinism may be introduced by many factors, such as the road conditions, driving situation, and nature of other vehicles.

Moreover, computing resources used inside an advanced vehicle are limited, and may fail. A dynamic and adaptive scheduling strategy may lead to a better utilization of resources and to select the important services regarding the system configuration.

The idea in chapter 5 is to consider a feedback-based scheduling scheme for driving assistance functions. The case study that we have considered for this purpose consists of a set of driving situations where each driving situation includes a set of metrics.

The feedback scheduling technique allows the distributed scheduling of tasks/messages regarding the driving situation change. Since the driving situation changes dynamically, the associated metrics and their components criticalness values change too. By applying the CDP-BBA scheduling strategy developed in chapter 4, we have re-scheduled the metrics components in a distributed environment while taking into account their criticalness values.

In order to use the adaptive scheduling techniques developed in this thesis, we plan to test them on real case study. For this purpose, we will consider the driving situations database of the CASSISE project. The aim of the CASSICE project (French acronym for Symbolic Characterization of Driving Situations) was to build an automatic classifier of driving situations. ADAS functions will be re-scheduled and adapted to the current vehicle situation and driver behavior using our proposed feedback scheduling techniques.

On the other hand, Vehicular Networks are a cornerstone of the envisioned Intelligent Transportation Systems (ITS). By enabling vehicles to communicate with each other via Inter-Vehicle Communication (IVC) as well as with roadside, vehicular networks will contribute to safer and more efficient roads by providing timely information to drivers and concerned authorities.

Thus, an important direction of perspective is the scheduling of cooperative ADAS in a distributed environment. Our laboratory is involved in two European Integrated Projects for the IVC technology such as SAFESPOT, and CVIS (2006-2010). These IVC activities motivate us to reconsider our framework SCOOT-R to deal with wireless environment. The idea is to design a *SCOOT-R Wireless* version that allows reliable communication and adaptive scheduling of messages between vehicles. Because of the limited precision and reliability of exchanged messages, the true values of the logical data are usually not known with certainty. The messages are thus considered with some level of confidence (that may be computed by a belief function, statistical measures, etc.).

# *Bibliography*


[AAS97] Tarek F. Abdelzaher, Ella M. Atkins, and Kang G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, 1997.

[AB90] N. Audsley and A. Burns. Real-time system scheduling. Technical report, University of York, 1990.

[AB98] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, Spain, 1998. IEEE Comput.

[AB99] L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.

[ABA+97] T. Abdelzaher, M. Bjorklund, S. Awson, W.-C. Feng, F. Jahanian, S. Johnson, P. Arron, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou. ARMADA middleware and communication services. In *proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, California, USA, 1997.

[ABD+95] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed priority preemptive scheduling: an historical perspective. *Journal of Real-Time Systems*, 8(2-3):173 – 198, 1995.

[ABRW91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, 1991.

[AC86] Steven Anderson and Marina C. Chen. Parallel branch-and-bound algorithms on the hypercube. In *second Conference on Hypercube Multiprocessors*, pages 309 – 317, Knoxville, Tennessee, Oakridge National Laboratories, September 1986.

[ACCP98] Emmanuelle Anceaume, Gilbert Cabillic, Pascal Chevochot, and Isabelle Puaut. HADES: A middleware support for distributed safety-critical real-time applications. In *proceedings of the International Conference on Distributed Computing Systems*, pages 344 – 351, 1998.

[Ack97] Sven Ackmer. Distributed real-time systems for automotive. Technical report, Computer Systems, Engineering Halmstad University, January 1997.

[AFH+03] J. Axelsson, J. Fröberg, H. Hansson, C. Norström, K. Sandström, and B. Villing. A comparative case study of distributed network architectures for different automotive applications. Technical Report 478, Mälardalen Research and Technology Centre, Department of Computer Science and Electronics, Mälardalen University, January 2003.

[AHCÅ05] Martin Andersson, Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. Simulation of wireless networked control systems. In *proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC 2005*, Seville, Spain, December 2005.

[And98] D. Anderson. *FireWire System Architecture* IEEE-1394. MinDShare, Inc, 1998.

[arc] Projet arcos: Action de recherche pour une conduite sécurisée.

[AS99] Tarek F. Abdelzaher and Kang G. Shin. QoS provisioning with qContracts in web and multimedia servers. In *proceedings of the IEEE Real-Time Systems Symposium (RTSS'99)*, pages 44–53, 1999.

[AWPvS01] João Paulo A. Almeida, Maarten Wegdam, Luís Ferreira Pires, and Marten van Sinderen. An approach to dynamic reconfiguration of distributed systems based on object-middleware. In *proceedings of 19th Brazilian Symposium on Computer Networks, (SBRC 2001)*, Santa Catarina, Brazil, May 2001.

[Bak03] David E. Bakken. *Middleware.* Kluwer Academic Press, 2003.

[Bat98] J. Bates. The state of the art in distributed and dependable computing. Technical Report 90-038, Laboratory for Communications Engineering, Cambridge University, October 1998.

[BBH+90] P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, D. Seaton, N. Speirs, and P. Veríssimo. The Delta-4 extra performance architecture (XPA). In *Digest of Papers of the 20th IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, pages 481 – 488, June 1990.

[BBL01] Guillem Bernat, Alan Burns, and Albert Llamosi. Weakly hard real-time systems. *IEEE transactions on computers*, 50(4):308 – 321, 2001.

[BC05] Olivier Bezet and Veronique Cherfaoui. On-line timestamping synchronization in distributed sensor architectures. In *proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 396 – 404, Washington, DC, USA, 2005.

[Ber93] Philip A. Bernstein. Middleware: An architecture for distributed system services. Technical Report CRL 93/6, Cambridge MA (USA), 1993.

[Bez05] Olivier Bezet. *Etude de la qualité temporelle des données dans un système distribué pour la fusion multi-capteurs.* Phd dissertation, Université de Technologie de Compiègne, France, 2005.

[BG92] Thomas E. Bihari and Prabha Gopinath. Object-oriented real-time systems: Concepts and examples. *Computer*, 25(12):25 – 32, 1992.

[BGM+97] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *proceedings of the 5th International Workshop on Quality of Service*, New York, June 1997.

[BISZ98] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *proceedings of the International Conference on Configurable Distributed Systems*, 1998.

[BLA98] G. C. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *proceedings of the IEEE Real-Time Systems Symposium*, pages 286 – 295, Washington, DC, USA, 1998.

[Bon99] Christian Bonnet. *Introduction aux systèmes temps-réel.* Hermes Editions, 1999.

[BPB00] A. Burns, S. Poledna, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49, 2000.

[BS89] T.P. Baker and A.C. Shaw. The cyclic executive model and Ada. *Journal of Real-Time Systems*, 1, 1989.

[Bur95] Alan Burns. Preemptive priority-based scheduling: an appropriate engineering approach. pages 225 – 248, 1995.

[Bus93] CAN Bus. Road vehicles - interchange of digital information - Controller Area Network (CAN) for high-speed communication, November 1993.

[But05] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer, New York, USA, 2005.

[BW96] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, 2nd edition, 1996.

[C.96] Shen C. On ATM support for distributed real-time applications. In *proceedings of IEEE Real-Time Technology and Applications Symposium*, Boston, MA, USA, June 1996.

[CB97] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *proceedings of the 18th IEEE Real-Time Systems Symposium*, page 330, Washington, DC, USA, 1997.

[CCS03a] K. Chaaban, P. Crubillé, and M. Shawky. Real-time embedded architecture for intelligent vehicles. In *proceedings of the Fifth Real-Time Linux Workshop*, Valencia, Spain, November 2003.

[CCS03b] K. Chaaban, P. Crubillé, and M. Shawky. SCOOT-R: A framework for distributed real-time applications. In *proceedings of the WIP of the 24h IEEE Real-Time Systems Symposium*, Cancun, Mexique, November 2003.

[CCS03c] K. Chaaban, P. Crubillé, and M. Shawky. SCOOT-R: Middleware communication services for real-time systems. In *proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS'03)*, pages 96 – 107, La Martinique, French West Indies, December 2003.

[CCS04] K. Chaaban, P. Crubillé, and M. Shawky. Real-time framework for distributed embedded systems. In *Principles of Distributed Systems*, volume 3144 of *Lecture Notes in Computer Science*, pages 96 – 107. Springer-Verlag GmbH, January 2004.

[CDD96] Flaviu Cristian, Bob Dancey, and Jon Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computer Systems*, 14(3):265 – 286, 1996.

[CDKM99] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *L'ordonnancement centralisé en temps réel.* Techniques de l'Ingénieur, Traité Informatique Industrielle, 1999.

[CDKM00] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *L'ordonnancement réparti en temps réel.* Techniques de l'Ingénieur, Traité Informatique Industrielle, 2000.

[CE00] A. Cervin and J. Eker. Feedback scheduling of control tasks. In *proceedings of the IEEE Conference on Decision and Control*, pages 4871 – 4876, Sydney, December 2000.

[Che99] Pascal Chevochot. *Conception de systèmes distribués temps-réel strict tolérants aux fautes.* Phd dissertation, Université de Rennes I, Rennes, France, January 1999.

[CHY+97] Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, and Yi-Min Wang. DCOM and CORBA side by side, step by step and layer by layer. 1997.

[CLW91] Lim C., Yao L., and Zhao W. A comparative study of three token ring protocols for real-time communications. In *proceedings of the 11th IEEE International Conference on Distributed Computing Systems*, pages 308 – 317, Arlington, Texas, USA, May 1991.

[CM95] C. Cardeira and Z. Mammeri. A schedulability analysis of tasks and network traffic in distributed real-times systems. *Journal of Measurement*, 15, 1995.

[CP99] P. Chevochot and I. Puaut. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies. In *proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Hong-Kong, China, December 1999.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249 – 274, 2000.

[CP01a] Pascal Chevochot and Isabelle Puaut. Conception de systèmes distribués temps-réel strict tolérants aux fautes avec du matériel sur étagère. In *proceedings of the 9th International Conference on Real-Time Systems (RTS'01)*, pages 209 – 226, Paris, Mars 2001.

[CP01b] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based WCET analysis. In *proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, page 37, Washington, DC, USA, 2001.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56 – 78, 1991.

[CRTM99] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communications. Technical report, Volvo, 1999.

[CS01] Junchul Chun and Jaegi Son. A CORBA-based telemedicine system for medical image analysis andmodeling. In *proceedings of the 14th IEEE Symposium on Computer-Based Medical Systems, CBMS'01*, pages 14 – 29, Bethesda, MD, USA, 2001.

[CSC04] K. Chaaban, M. Shawky, and P. Crubillé. Dynamic reconfiguration for high level in-vehicle applications using IEEE-1394. In *proceedings of the IEEE Conference on Intelligent Transportation Systems (ITSC)*, Washington, D.C, October 2004.

[CSC05] K. Chaaban, M. Shawky, and P. Crubillé. A distributed framework for real-time in-vehicle applications. In *proceedings of the IEEE Conference on Intelligent Transportation Systems ITSC*, Vienna, Austria, September 2005.

[CSR89] S.-C. Cheng, J.-A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems: a brief survey. *Tutorial: hard real-time systems*, pages 150 – 173, 1989.

[CW96] Geoff Coulson and Daniel Waddington. A CORBA compliant real-time multimedia platform for broadband networks. In *proceedings of the International Workshop on trends in distributed systems (TreDS)*, volume 1161, pages 14 – 29, 1996.

[CZ95] Cardeira C. and Mammeri Z. A schedulability analysis of tasks and network traffic in distributed real-time systems. *Journal of Measurement*, 15(2):71 – 83, 1995.

[DA96] Sekhar Darbha and Dharma P. Agrawal. Scalable scheduling algorithm for distributed memory machines. In *proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP'96)*, page 84, Washington, DC, USA, 1996.

[dC96] Francisco VASQUES de CARVALHO. *Sur l'intégration de mécanismes d'ordonnancement et de communication dans la sous-couche MAC de réseaux locaux temps-réel.* Phd dissertation, Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS), Toulouse, June 1996.

[dOdSF00] R. Silva de Oliveira and J. da Silva Fraga. Fixed priority scheduling of tasks with arbitrary precedence constraints in distributed hard real-time systems. *Journal of Systems Architecture*, 46, 2000.

[DTT99] Anne-Marie Déplanche, Pierre-Yves Théaudière, and Yvon Trinquet. Implementing a semi-active replication strategy in CHORUS/ClassiX, a distributed real-time executive. In *Symposium on Reliable Distributed Systems*, pages 90 – 101, 1999.

[ea93] Agrawal G. et al. Local synchronous capacity allocation schemes for guaranteeing messages deadlines with the timed token protocol. In *proceedings of INFOCOM'93*, pages 186 – 193, San Francisco, 1993.

[EC886] Expert Choice. McLean, VA: Decision support software, 1986.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114 – 131, 2003.

[Emm00] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *proceedings of the Conference on The Future of Software Engineering, (ICSE'00)*, pages 117 – 129, New York, NY, USA, 2000.

[Fer93] Edward E. Ferguson. Resource scheduling for adaptive systems. In *proceedings of the IEEE Workshop on Real-Time Applications*, pages 102 – 103, New York, USA, May 1993.

[Gei01] Kurt Geihs. Middleware challenges ahead. *Computer*, 34(6):24 – 31, 2001.

[GEP+00] Juan C. Guerri, Manuel Esteve, Carlos Palau, Manuel Monfort, and M. Angeles Sarti. A software tool to acquire, synchronize and playback multimedia data: an application in kinesiology. *Computer Methods and Programs in Biomedicine*, 62, 2000.

[GFS93] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. A testbed for optimistic execution of realŋtime simulations. In *proceedings of the IEEE Workshop on Parallel and Distributed RealŋTime Systems*, April 1993.

[GGM97] Jean Marc Geib, Christophe Gransart, and Philippe Merle. *CORBA: des concepts à la pratique*. Inter Editions, Kraig Brocksmith, 1997.

[Gho94] K. Ghosh. A survey of real-time operating systems. Technical Report GIT-CC-93/18, Georgia Institute of Technology, Atlanta, Georgia, February 1994.

[GLS01] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA scheduling service. *Journal of Real-Time Systems*, 20(2):117 – 154, 2001.

[GNSC04] Aniruddha S. Gokhale, Balachandran Natarajan, Douglas C. Schmidt, and Joseph K. Cross. Towards real-time fault-tolerant CORBA middleware. *Cluster Computing*, 7(4):331 – 346, 2004.

[Gro00] Most Group. Digitization opens the way for new standard. Technical report, Automobilentwicklung, 2000.

[H03] Kopetz H. Time-triggered real-time computing. *Annual Reviews in Control*, 27:3 – 13(11), 2003.

[HCr03] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. TrueTime: Real-time control system simulation with MATLAB/Simulink. In *proceedings of the Nordic MATLAB Conference*, Copenhagen, Denmark, October 2003.

[HLB+97] Hans Hansson, Harold Lawson, Olof Bridal, Christer Eriksson, Sven Larsson, Henrik L., and Mikael S. BASEMENT: An architecture and methodology for distributed automotive real-time systems. *IEEE Transactions on Computers*, 46(9):1016 – 1027, 1997.

[HRP+93] Klein M. H, T. Ralya, B. Pollak, R. Obenza, and M. G Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[IN02] Damir Isovic and Christer Norström. Components in real-time systems. In *proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, March 2002.

[Jef92] Kevin Jeffay. On kernel support for real-time multimedia applications. In *proceedings of the Third IEEE Workshop on Workstation Operating Systems*, pages 39 – 46, Key Biscayne, FL, April 1992.

[JST99] Stankovic J.A., Chenyang Lu Son, and S.H. Gang Tao. The case for feedback control real-time scheduling. In *proceedings of the 11th Euromicro Conference*, York ,UKA, 1999.

[Jun] RPC: Remote procedure call protocol specification, version 2.

[JYlM+01] P. Johansson, E. YZ, l Mario, G. Manthos, and K. UCLA. Bluetooth an enabler of personal area networking. *IEEE Network, Special Issue in Personal Area Networks*, 2001.

[KBM04] J. Kaiser, C. Brudna, and C. Mitidieri. COSMIC: a real-time event-based middleware for the CAN-bus. *Journal of Systems and Software (JSS)*, 2004.

[KDK+89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro.*, 9(1):25 – 40, 1989.

[KG94] Hermann Kopetz and Gunter Grunsteidl. TTp-A protocol for fault-tolerant real-time systems. *IEEE transactions on Computer*, 27(1):14 – 23, 1994.

[KKMS95] H. Kopetz, A. Kruger, D. Millinger, and A. Schedl. A synchronization strategy for a time-triggered multicluster real-time system. In *proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, Bad Neuenahr, Germany, September 1995.

[KM85] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424 – 436, 1985.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293 – 1306, 1990.

[KO87] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933 – 940, 1987.

[Kop97a] H. Kopetz. Components-based design of large distributed real-time systems. In *proceedings of The 14th IFAC Workshop on Distributed Computer Control Systems*, pages 171 – 177, Seoul, Korea, 1997.

[Kop97b] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[Kop98] H. Kopetz. The Time-Triggered Architecture. In *proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98)*, page 22, Washington, DC, USA, 1998.

[Kop00] Hermann Kopetz. A comparison of TTP/C and FlexRay. Research Report 22/2000, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2000.

[LA99] H. Lonn and J. Axelsson. A comparison of fixed-priority and static cyclic scheduling for distributed automotive control applications, 1999.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46 – 61, 1973.

[LMS04] S. Lin, G. Manimaran, and BL. Steward. Feedback-based real-time scheduling in autonomous vehicle systems. In *proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 316, Washington, DC, USA, 2004.

[LN99] B. Li and K. Nahrstedt. A control-based middleware framework for Quality of Service adaptations. *IEEE Journal on Select. Areas Commun.*, 1999.

[LRWK03] Peng Li, Binoy Ravindran, Jinggang Wang, and Glenn Konowicz. Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation. In *proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, Hakodate, Hokkaido, Japan, 2003.

[LRWK04] P. Li, B. Ravindran, J. Wang, and G. Konowicz. Fast, best-effort real-time scheduling algorithms. *IEEE Transactions on computers*, 53, September 2004.

[MGFSK04] Klaus D. Müller-Glaser, Gerd Frick, Eric Sax, and Markus Kühl. Multiparadigm modeling in embedded systems design. *IEEE Transactions on control systems technology*, 12, 2004.

[Mil81] Leslie Jill Miller. The ISO reference model of open systems interconnection: A first tutorial. In *proceedings of the ACM '81 conference*, pages 283 – 288, New York, NY, USA, 1981.

[MMG04] D. Marinca, P. Minet, and L. George. Analysis of deadline assignment methods in distributed real-time systems. *Computer Communications*, 27(15), 2004.

[MMM98] A. Mittal, G. Manimaran, and C.S.R Murthy. Integrated dynamic scheduling of hard and QoS degradable real-time tasks in multiprocessor systems. In *proceedings of the Fifth*

*International Conference on Real-Time Computing Systems and Applications*, pages 127 – 136, Hiroshima, Japan, 1998.

[MMM00] Anita Mittal, G. Manimaran, and C. Siva Ram Murthy. Integrated dynamic scheduling of hard and QoS degradable real-time tasks in multiprocessor systems. *Journal of System Architecture*, 46(9):793 – 807, 2000.

[MMMM01] G. Manimaran, Shashidhar Merugu, Anand Manikutty, and C. Siva Ram Murthy. Integrated scheduling of tasks and messages in distributed real-time systems. *Engineering of distributed control systems*, pages 99 – 112, 2001.

[Mok83] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[MSP00] Shawky M., Favard S., and Crubillé P. A computing platform and its tools for feature extraction from on-vehicle image sequences. In *proceedings of the 3rd IEEE Annual conference on Intelligent Transportation Systems (ITSC)*, Dearborn, Michigan, USA, Octobre 2000.

[MZ95] N. Malcolm and W. Zhao. Hard real-time communication in multiple-access networks. *Journal of Real-Time Systems*, 8:35 – 77, 1995.

[Nas00] F. Nashashibi. RTm@ps: a framework for prototyping automatic multi-sensor applications. In *proceedings of The IEEE Intelligent Vehicles Symposium*, Dearborn, Michigan, USA, October 2000.

[NB02] M. EL Najjar and P. Bonnifait. A road reduction method using multi-criteria fusion. In *proceedings of The IEEE Intelligent Vehicles Symposium*, Versailles, France, June 2002.

[NB05] Maan E. El Najjar and Philippe Bonnifait. A road-matching method for precise vehicle localization using belief theory and kalman filtering. *Journal of Autonomous Robots*, 19(2):173 – 191, 2005.

[NGYS00] Balachandran Natarajan, Aniruddha S. Gokhale, Shalini Yajnik, and Douglas C. Schmidt. DOORS: Towards high-performance fault tolerant CORBA. In *proceedings of the International Symposium on Distributed Objects and Applications*, pages 39 – 48, 2000.

[Nic98] Guillem Bernat Nicolau. *Specification and Analysis of Weakly Hard Real-Time Systems*. Phd dissertation, Universitat de les Illes Balears, Departament de Ciències Matemàtiques i Informàtica, Spain, January 1998.

[NTG02] Takashi Norimatsu, Hideaki Takagi, and H. Richard Gail. Performance analysis of the IEEE 1394 serial bus. *Journal of Performance Evaluation*, 50(1-4):1 – 26, 2002.

[OSK+00] Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons, Irfan Pyarali, and David L. Levine. Evaluating policies and mechanisms for supporting embedded, real-time applications with CORBA 3.0. In *proceedings of Sixth IEEE Real Time Technology and Applications Symposium (RTAS'00)*, page 188, WashingtonD.C., USA, 2000.

[PB00] D. Prasad and A. Burns. A value-based scheduling approach for real-time autonomous vehicle control. *Robotica*, 18(3):273 – 279, 2000.

[PD93] F. Panzieri and R. Davoli. Real-time systems: a tutorial. *Performance evaluation of computer and communication systems*, 729:435 – 462, 1993.

[PG96] Bonnifait P. and Garcia G. A multisensor localization algorithm for mobile robots and its real-time experimental validation. In *proceedings of the IEEE International Conference on Robotics and Automation (ICRA'96)*, pages 1395 – 1400, Minneapolis, Minnesota, USA, 1996.

[PG98] Bonnifait P. and Garcia G. Design and experimental validation of an odometric and goniometric localization system for outdoor robot vehicles. *IEEE Transactions on Robotics and Automation*, 14(4):541 – 548, 1998.

[PJ90] Dan Poirier and Kevin Jeffay. An implementation and application of the real-time producer/consumer paradigm. Technical Report 90-038, University of North Carolina at Chapel Hill Department of Computer Science, January 1990.

[PPPD01] Bonnifait P., Bouron P., Crubille P., and Meizel D. Data fusion of four ABS sensors and GPS for an enhanced localization of car-like vehicles. In *proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1050 – 4729, Washington, D.C, 2001.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency - Practice and Experience*, 9(11):1225 – 1242, 1997.

[Raj98] Gopalan Suresh Raj. A detailed comparison of CORBA, DCOM and Java/RMI, September 1998.

[Ram87] Krithi Ramamritham. Channel characteristics in local-area hard real-time systems. *Computer Network and ISDN Systems*, 13(1):3 – 13, 1987.

[RE97] Mark Roy and Alan Edward. Inside DCOM: Microsoft's distributed object architecture extends the capabilities of COM to work across the network. *DBMS Journal*, 10(4):26 – 34, 1997.

[RHS97] Minsoo Ryu, Seongsoo Hong, and M. Saksena. Streamlining real-time controller design: From performance specifications to end-to-end timing constraints. In *proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, Washington, DC, USA, 1997.

[roa] European project roadsense: Road awareness for driving via a strategy that evaluates numerous systems.

[rpc] RPC: Remote procedure call protocol specification, version 2, june 1988.

[RS01] John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, Washington, DC, USA, 2001.

[RSZ89a] K. Ramamritham, A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38, 1989.

[RSZ89b] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110 – 1123, 1989.

[Sch98] Douglas C. Schmidt. An architectural overview of the ACE framework, 1998.

[Sch02] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43 – 48, 2002.

[SGB87] K. Schawn, P. Gopinath, and W. Bo. CHAOS- kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, 36(8):904 – 916, 1987.

[SGG+99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *proceedings of the third symposium on Operating systems design and implementation (OSDI '99)*, pages 145 – 158, Berkeley, CA, USA, 1999.

[SGHP97] Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison, and Guru Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Communications Magazine*, 14, 1997.

[SJK88] Biyabani S.R., Stankovic J.A., and Ramamritham K. The integration of deadline and criticalness in hard real-time scheduling. In *proceedings of the Real-Time Systems Symposium*, pages 152 – 160, Huntsville, AL, USA, 1988.

[SK91a] David B. Stewart and Pradeep K. Khosla. Real-time scheduling of dynamically reconfigurable systems. In *proceedings of the IEEE International Conference on Systems Engineering*, pages 139 – 142, Dayton Ohio, August 1991.

[SK91b] David B. Stewart and Pradeep K. Khosla. Real-time scheduling of sensor-based control systems. In *proceedings of Eighth IEEE Workshop on Real-Time Operating systems and Software*, pages 144 – 150, Atlanta, GA, 1991.

[SK96] Kweon S.K and Shin K.G. Traffic-controlled rate monotonic priority scheduling of ATM cells. In *proceedings of 15th IEEE INFOCOM conference*, San Francisco, USA, March 1996.

[SK00] Douglas C. Schmidt and Fred Kuhns. An overview of the real-time CORBA specification. *Computer*, 33(6):56 – 63, 2000.

[SLSS96] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 13 – 21, Washington, DC, USA, 1996.

[SM99] Kang G. Shin and Charles L. Meissner. Adaptation of control system performance by task reallocation and period modification. In *proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS99)*, York, England, 1999.

[SM03] S. M. Sadjadi and P. K. McKinley. A survey of adaptive middleware. Technical report, Computer Science and Engineering, Michigan State University, 2003.

[SMS90] S. Som, R. Mielke, and W. Stoughton. Strategies for predictability in real-time data-flow architectures. In *proceedings of the 11th IEEE Real-Time Systems Symposium*, page 226 Ű 237, Florida, USA, 1990.

[SO03] Douglas C. Schmidt and Carlos O'Ryan. Patterns and performance of distributed real-time and embedded publisher/subscriber architectures. *Journal of Systems Software*, 66(3):213 – 223, 2003.

[SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Journal of Real-Time Systems*, 2, 1990.

[SR91] John A. Stankovic and Krithi Ramamritham. The Spring kernel: A new paradigm for real-time systems. *IEEE Software Magazine*, 8(3):62 – 72, 1991.

[SRF03] C. Steger, P. Radosavljevic, and P. Frantz. Performance of IEEE 802.11b wireless LAN in an emulated mobile channel. In *proceedings of the IEEE Vehicular Technology Conference (VTC)*, Jeju, Korea, April 2003.

[SSR03] O Sename, D. Simon, and D. Robert. Feedback scheduling for real-time control of systems with communication delays. In *proceedings of the IEEE conference on the Emerging Technologies and Factory Automation (ETFA '03)*, pages 454 – 461, Lisbon, Portugal, 2003.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Transactions on Computers*, 21(10):10 – 19, 1988.

[Sta96] John A. Stankovic. Realŋtime and embedded systems. *Robotica Journal*, 1996.

[Ste01] B. Steux. *RTMAPS, un environnement logiciel dédié à la conception d'applications embarquées temps-réel. Utilisation pour la détection automatique de véhicules par fusion radar/Vision.* Phd dissertation, Ecole des mines de Paris, Paris, December 2001.

[TC94] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess, Microprogram.*, 40(2-3):117 – 134, 1994.

[Tin93] K. Tindell. *Fixed priority scheduling of hard real-time systems.* Phd dissertation, Department of Computer Science,University of York, UK, 1993.

[TM89] H. Tokuda and C. W. Mercer. ARTS: a distributed real-time kernel. *SIGOPS Operating Systems Review*, 23(3):29 – 53, 1989.

[TN91] Hideyuki Tokuda and Tatsuo Nakajima. Evaluation of real-time synchronization in real-time Mach. In *proceedings for the USENIX 1991 Mach Workshop*, pages 213 – 222, October 1991.

[Tri03] Y. Trinquet. Noyaux temps-réel: le cas de osek-vdx, *École d'été Temps Réel (ETR2003), Toulouse, 9 - 12 September*, 2003.

[TW] A. Tindell, K.and Burns and A. Wellings. Calculating controller area network (CAN) message response times. In *Distributed Computer Control Systems (DCCS'94). IFAC Workshop*. Pergamon; Oxford, UK.

[VB94] Berge-Cherfaoui V. and Vachon B. Dynamic configuration of mobile robot perceptual system. In *proceedings of the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI'94)*, pages 707 – 714, Washington DC, USA, 1994.

[Vin93] Steve Vinoski. Distributed object computing with corba. *C++ Report Magazine*, 1993.

[Vin02] Steve Vinoski. Where is middleware? *IEEE Internet Computing*, 6(2):83 – 85, 2002.

[VZF91] D.C. Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communications in a packet-switched network. In *proceedings of IEEE TRICOM'91*, pages 35 – 43, April 1991.

[Yag01] K. Yaghmour. The real-time application interface. In *proceedings of the* L*inux Symposium*, Ottawa, Canada, July 2001.

[YL05] S.O. Yahia and S. Loriette. Fuzzy querying of evolutive situations: Application to driving situations. *Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII)*, 9, 2005.

[ZH03] Armin Zimmermann and Günter Hommel. A train control system case study in model-based real time system design. In *proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2003.

[ZPS+99] K. Zuberi, P. Pillai, K. Shin, T. Imai, W. Nagaura, and S. Suzuki. EMERALDS-OSEK: A small real-time operating system for automotive control and monitoring. In *proceedings of the SAE International Congress*, Indianapolis, IN, March 1999.

**Titre :**
Architecture Informatique Temps-Réel Pour Véhicules Avancés

**Résumé :**
Cette thèse se situe dans le domaine des systèmes informatiques temps-réel embarqués, plus particulièrement les logiciels embarqués dans l'automobile pour tous les dispositifs émergents et à venir d'évaluation des systèmes d'aide à la conduite (ADAS) pour les prochaines générations de véhicules. Ce document présente les trois axes principaux des travaux de cette thèse : Le premier axe comprend le développement d'un middleware reconfigurable dynamiquement, SCOOT-R. Le deuxième axe concerne le développement des techniques d'ordonnancement distribuées des opérations SCOOT-R avec un objectif de qualité de service de bout en bout. Finalement, le développement des techniques d'ordonnancement régulé pour l'adaptation du système à des situations de conduite et comportements du conducteur variables. Dans ce cas l'importance des fonctions est adaptée suivant le contexte momentané du système.

**Mots-clés :**
Systèmes distribués temps-réel, Intergiciels temps-réel, Ordonnancement temps-réel, Ordonnancement contextuel, Applications automobiles

**Title :**
A Distributed Real-Time Architecture For Advanced Vehicles

**Abstract:**
This thesis falls in the field of embedded real-time systems, and more precisely the in-vehicle embedded software for the evaluation of the next generation of driving assistance systems (ADAS). This document presents the three principal axes of the thesis: First, the development of a dynamic reconfigurable middleware called SCOOT-R. Second, the development of distributed real-time scheduling strategies in order to schedule SCOOT-R operations with the main goal of end-to-end QoS guarantee. Finally, the development of feedback-based scheduling schemes to schedule driving assistance systems. The adaptation in this scheme is carried out according to the current driving situation and the driver behavior, which will further lead to the change of the associated driving assistance function's criticalness.

**Keyword :**
Distributed real-time systems, (OO) and real-time middleware, Real-time scheduling, Feedback scheduling, Automotive applications