A Novel Software Framework for Embedded Multiprocessor Smart Cameras

ANDREAS DOBLANDER
Allgemeines Rechenzentrum GmbH
ANDREAS ZOUFAL
Austrian Research Centers GmbH
and
BERNHARD RINNER
Klagenfurt University

Distributed smart cameras (DSC) are an emerging technology for a broad range of important applications including smart rooms, surveillance, entertainment, tracking, and motion analysis. By having access to many views and through cooperation among the individual cameras, these DSCs have the potential to realize many more complex and challenging applications than single-camera systems.

This article focuses on the system-level software required for efficient streaming applications on single smart cameras as well as on networks of DSCs. Embedded platforms with limited resources do not provide middleware services well known on general-purpose platforms. Our software framework supports transparent intra- and interprocessor communication while keeping the memory and computation overhead very low. The software framework is based on a publisher–subscriber architecture and provides mechanisms for dynamically loading and unloading software components as well as for graceful degradation in case of software- and hardware-related faults. The software framework has been completely implemented and tested on our embedded smart cameras consisting of an ARM-based network processor and several digital signal processors. Two case studies demonstrate the feasibility of our approach.

Categories and Subject Descriptors: D.2.11 [**Domain-specific architectures**]: Patterns; C.3 [**Real-time and embedded systems**]

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Smart cameras, publisher–subscriber, fault tolerance, video surveillance, distributed embedded systems

The authors would like to acknowledge support from Texas Instruments.

Authors' addresses: A. Doblander, Allgemeines Rechenzentrum GmbH, Tschamlerstraße 2, A-6020 Innsbruck, Austria; email: andreas.doblander@arz.at; A. Zoufal, Austrian Research Centers GmbH, Donau-City-Straße 1, A-1220 Wien, Austria; email: andreas.zoufal@arcs.ac.at; B. Rinner, Institute of Networked and Embedded Systems, Klagenfurt University, Lakeside B02b, A-9020 Klagenfurt, Austria; email: bernhard.rinner@uni-klu.ac.at.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1539-9087/2009/04-ART24 $5.00
DOI 10.1145/1509288.1509296 http://doi.acm.org/10.1145/1509288.1509296

ACM Reference Format:
Doblander, A., Zoufal, A., Rinner, B. 2009. A novel software framework for embedded multiprocessor smart cameras. ACM Trans. Embedd. Comput. Syst. 8, 3, Article 24 (April 2009), 30 pages. DOI = 10.1145/1509288.1509296 http://doi.acm.org/10.1145/1509288.1509296

1. INTRODUCTION

Recently, much effort has been put into the development of distributed vision systems with smart cameras [Wolf et al. 2002; Bramberger et al. 2006] as key components. Smart cameras combine video sensing, processing, and communication within a single embedded device and are equipped with a high-performance onboard computing and communication infrastructure. Instead of streaming raw video data they typically deliver abstracted information such as color or geometric features, segmented objects, or rather high-level decisions from the observed scene.

Networks of distributed smart cameras (DSC) [Aghajan and Kleihorst 2007; Rinner and Wolf 2008a] are an emerging technology for a broad range of important applications including smart rooms, surveillance, entertainment, tracking, and motion analysis. By having access to many views and through cooperation among the individual cameras, these networks have the potential to realize many more complex and challenging applications than single-camera systems. DSCs exemplify two recent trends in visual computing research: distributed processing and embedded computing. Thus, DSC systems use distributed algorithms to perform complex vision tasks across multiple cameras in real-time.

Designing, implementing, and deploying applications on DSC networks is much more complex than for single-camera systems. On general-purpose platforms, distributed applications are often developed based on a middleware system, which provides services for networking and data transfer. On DSC networks, we would like to take advantage of middleware services as well. However, the requirements of a middleware, for distributed image processing on embedded devices are significantly different. Component-based middleware, such as DCOM or CORBA, are targeted for general-purpose computing and are not suitable for resource-limited devices. The CORBA technology has been adapted to resource-constrained real-time systems (e.g., by the Real-Time CORBA [RT-CORBA] specification and its TAO implementation Schmidt [2002]). However, this approach is still very resource consuming. On the other hand, recent research in wireless sensor networks (WSN) has come up with some interesting middleware concepts as well [Akyildiz et al. 2002]. Due to the nature of WSNs, these middleware systems especially focus on reliable services for ad-hoc networks and energy awareness [Molla and Ahamed 2006].

DSC networks differ from WSNs in various aspects as well. First, the amount of data to be processed is much higher in DSC networks than in WSNs. Second, individual processing nodes in a DSC network are more capable than in WSNs. While resource constraints on the embedded smart cameras are important, the resource limitations, especially energy, are of top priority in WSN. Third, due to ad-hoc networking, communication in WSN has a very dynamic nature. DSCs,

on the other hand, are typically connected via wired networks providing higher communication bandwidths.

This article focuses on the system-level software required for efficient video processing applications on single smart cameras as well as on networks of DSCs. We describe the architecture of this middleware, discuss the design and implementation alternatives, present performance data, and demonstrate its applicability towards fault tolerance. The main contributions of this research can be summarized as follows:

Software framework for embedded multiprocessor platforms. The developed software framework is based on a publisher–subscriber (PS) model designed for real-time multimedia applications. This software framework supports transparent intra- and interprocessor communication and scales well with the number of processors on the embedded platform. Our software framework introduces very little overhead concerning memory requirements and communications times compared to an implementation using standard operating system calls.

Dynamic component composition. Our software framework supports dynamic component composition—a feature typically known only on general-purpose platforms. Algorithms can be specified using a component model, which includes the algorithm's binary, the resource requirements, the performance ratings, and the reconfigurable algorithm attributes. By monitoring the available system resources and by exploiting dynamic loading and reconfiguration, software components can be loaded and unloaded on the embedded platform during runtime. Thus, dynamic component composition provides the mechanism to change the functionality (code and quality of service [QoS]) on demand and during runtime.

Simple but effective fault-tolerance mechanisms. The software framework includes simple but effective mechanisms for fault tolerance as well. Various software- and hardware-related faults can be detected by monitoring the resource utilization and by applying simple fault detection mechanisms such as alive messaging and watchdog timers. By exploiting dynamic reconfiguration, detected faults can be eluded and the services may then still be available—potentially at a lower QoS-level.

Implementation and case studies. The software framework has been completely implemented on our embedded smart camera platform (SmartCam). Although our smart camera is a dedicated hardware platform consisting of a network processors and several digital signal processors (DSPs), the software framework can be easily ported to other platforms. Several case studies have been conducted to demonstrate the feasibility of our approach.

The remainder of this article is organized as follows. Section 2 reviews related work on middleware and frameworks as well as component models and technology. This review on related work focuses on embedded systems with their inherent strong resource limitations. Section 3 briefly summarizes our embedded multiprocessor SmartCam architecture, which provides the hardware and basic software platform for the implementation of our novel software

framework. Section 4 presents our middleware architecture in detail. We first describe the design of the PS architecture for single-camera applications and then focus on dynamic component composition based on our PS architecture. Section 5 describes our fault-tolerance approach for graceful degradation in networks of DSCs. Section 6 presents experimental results concerning the performance of the PS middleware as well as an evaluation of the fault-tolerance architecture using two fault scenarios. Section 7 concludes this article with a discussion and an outlook for future work.

2. RELATED WORK

2.1 Middleware and Frameworks for Embedded Systems

2.1.1 *CORBA-Based General-Purpose Middleware.* Middleware for distributed and embedded systems is a very active research field. Much work has been done to support transparent communication and to ease distributed application development. Component-based middleware technologies from general-purpose computing, such as, Microsoft DCOM [Sessions 1997], Java RMI [Pitt and McNiff 2001], and OMG CORBA [Pope 1998] are not suitable for very resource-limited devices [Mascolo et al. 2002]. To adapt the CORBA technology to resource-constrained real-time systems the Real-Time CORBA (RT-CORBA) and minimum CORBA specifications [The Object Management Group 2001; Object Management Group 2002] have been introduced.

Schmidt et al. [2002] invented "TAO" as an implementation of the RT-CORBA specification. It is an object request broker especially developed for distributed real-time and embedded systems. Their *CIAO* framework [Balasubramanian et al. 2003] extends TAO to also include a component model for distributed real-time and embedded systems that enables easy component composition. All these approaches are quite large and, therefore, not suitable for our multi-DSP platform. They are further not available on the operating system of our DSPs and cannot easily be ported to it.

In general, all these approaches share the idea of providing transparent communication among objects or components residing in different address spaces. The problem is that they also aim at supporting a wide range of programming languages and mostly general-purpose computer architectures. That is the reason why these middleware systems impose substantial overhead. What they provide rather well is software reuse and platform independence. But as these advantages cannot be exploited in the highly specialized SmartCam platform, a very light-weight approach was chosen and is presented in this work.

2.1.2 *A Microbroker-based Middleware for Pervasive Computing.* In Becker et al. [2003] the authors present their BASE middleware for pervasive computing. This article aims at a scalable and efficient middleware that serves all possible computing architectures for pervasive computing.

BASE is based on a microbroker that only implements very basic functionality. All other features can be added as plug-ins, as needed. Especially, transport protocols are added as plug-ins. By this technology, it is easy to adapt the middleware to new protocols and communication devices.

In this article a very similar approach is presented where a medium-abstraction entity takes care of transparent communication. This abstraction object is also easily extended to handle new communication media. The microbroker is quite analogous to micro kernels known from operating system technology.

Although the BASE middleware was implemented in Java, it is very memory efficient due to the microbroker approach that focuses only on the most important middleware features. However, transport of remote invocations is realized through the Java RMI interface. Therefore, it suffers from substantial performance overhead for remote service invocations. It exploits locality so that unlike RMI, it does not need to pass through the RMI and TCP protocol stacks.

2.1.3 *Distributed SW Architecture for Ubiquitous Sensor Systems.* Lin et al. [2006] present a software framework for ubiquitous smart cameras. It is a joint effort of the Princeton's smart camera group and the Vanderbilt University's Model-Integrated Computing (MIC) group. Their focus lies on the modeling and design of real-time embedded camera systems.

Based on their gesture recognition system prototype they investigate a fully distributed communication pattern to support intelligent and ubiquitous applications using several cameras. As the heart of the system, a multilayer software framework provides a service-oriented platform for different algorithms.

Although their goals are similar to ours, their implementation differs significantly. Their framework is based on the model-integrated computing (MIC) environment [Karsai et al. 2003], and they use DirectX as a middleware system. Standard PCs have been used as prototyping platforms, which is obviously not a distributed embedded system.

2.1.4 *Texas Instruments DaVinci Technology.* The *DaVinci* technology by Texas Instruments (TI) [Mody 2006] is an innovative framework for multicore embedded DSP solutions. The intended applications are multimedia appliances that rely strongly on complex signal processing algorithms. Extending previous architectures, TI provides a complete software bundle to ease application development. On the one hand there are the two operating systems, that is, Linux for the ARM and DSP/BIOS for the C64x DSP, along with different support libraries. On the other hand there is an abstraction to aid developers in using third-party components easily.

Similar to our approach presented in previous work [Bramberger et al. 2006], signal processing algorithms are treated as components. The application developer can plug and unplug them using standardized interfaces. But in contrast to our approach, they currently support only encoder and decoder algorithms. Based on their XDAIS [Instruments 2002] component standard, they extended it to XDAIS-DM or XDM to also support algorithm descriptions that are needed for proper composition of multimedia algorithms. Mainly, this information is dedicated to different QoS settings as resolution, frame rate, and the like.

In contrast, the algorithm description interface presented in this article is more flexible and is not limited to encoder and decoder tasks. Another difference to the presented approach is that the XDAIS-DM framework focuses on a single system-on-chip. Indeed, it handles two different cores, but it does not address

distributed nodes and communication among different algorithms residing in different address spaces as does the framework presented in this article.

2.2 Component Models and Technology

2.2.1 *Large-Scale Server Component Models.* One of the most well-known component models is the CORBA component model (CCM) [Group 2005] that is specified for the CORBA specification in its third version. Other well-known commercial component-based approaches include Sun *Enterprise Java Beans* [DeMichiel 2002] and Microsoft *.NET* [Microsoft 2005]. These are full-featured component systems that are mostly used in the development of large-scale business applications.

Because of the rich feature set, they are also very large software systems that also impose substantial performance overhead. In embedded systems, the focus is on light-weight solutions and, therefore, these major component systems along with their corresponding component models are not suitable in a typical embedded setting. To overcome the problems of excessive memory and computing power requirements, *Light-Weight CCM* (LwCCM) [Systems and Thales 2003] was submitted to the Object Management Group for specification.

LwCCM aims at providing only core features. Advanced functionality of the CCM is not included in LwCCM. Thus, it can be implemented for resource-critical embedded systems. Embedded CORBA-based applications can, therefore, be realized using LwCCM. Persistence, transactions, and security are not addressed in the LwCCM specification. Nevertheless, compatibility with the full-flagged CCM specification is retained so that LwCCM components can also be deployed on CCM-based systems.

2.2.2 *SaveCCM—A Component Model for Safety-Critical Real-Time Systems.* SaveCCM [Hansson et al. 2004] is a specialized component model aimed at safety-critical control applications in vehicular systems. It is only of limited flexibility but, on the other hand, facilitates analysis of real-time and dependability issues in embedded control systems. As part of an overall effort to improve dependability in vehicular systems, SaveCCM is also accompanied by a dedicated component framework to improve development processes.

Note that the term SaveCCM has nothing to do with the CCM. It is merely a composition of the project name *SAVE*, the framework SaveComp, and the general term component model and might be stated as SaveComp component model. Based on a pipes and filters paradigm, the execution model of SaveCCM is rather restrictive. Components as the basic unit of encapsulation can be in either state, executing or waiting to be triggered, respectively.

The component model defines three other entities besides a component. First, there are switches that are used to dynamically change component interconnections. Second, assemblies are a means for forming aggregate components. As the third part, the runtime framework provides services like component communication, component execution, and control of sensors and actuators.

2.2.3 *An Efficient Component Model for the Construction of Adaptive Middleware.* In Clarke et al. [2001], the authors present OpenCOM, which is a light-weight component model based on the standardized COM component

model [DeMichiel 1995]. To be efficient, it only supports a subset of the overall COM specification. That is, only a single address space is supported.

This is in contrast with the approach described in this article, where the benefit is that component interaction is supported beyond address space boundaries in a transparent way. Furthermore, OpenCOM does not implement standard component model features such as distribution, persistence, security, and transactions. But the OpenCOM model is designed for dynamic reconfiguration of components, which is in contrast to most standard component models that do not very well support the deployment phase of components in a dynamic application environment.

The interesting thing with OpenCOM is that it is designed as a component model for the design of middleware platforms itself. That is, it is not used to provide a structure for component interaction on top of a framework to form applications but to develop the framework.

2.2.4 *AFT-CCM—Adaptive Fault-Tolerance on the CORBA Component Model.* AFT-CCM [Fraga et al. 2003] is a component model based on CCM. It is aimed at applications with fault-tolerance requirements. Like most CORBA-based technologies it is also designed for large-scale distributed computing systems, mostly applied in Web applications. The application programmer can specify QoS requirements for services and the desired levels of dependability can also be defined.

To achieve a special dependability level, different forms of component (i.e., service) replication are employed. Several dedicated system components are responsible for the transparent replication of application components. Furthermore, key system components are also replicated on different hosts in the system to guarantee correct replication also in case of failures in the runtime environment supporting the component model. Of course, it is also possible to integrate components into the system that are not critical and, therefore, do not need to be replicated. Persistence of component state information is achieved by constantly saving it to local nonvolatile storage. Hence, on failure of a component, its state is restored to a replica, to continue normal operation after minimum downtime.

Given the significant overhead of the overall management framework and the full redundancy of replication, it is understandable that each host in such a system has to provide substantial hardware resources. Therefore, AFT-CCM is not suitable for cost-sensitive embedded applications.

3. EMBEDDED SMART CAMERA PLATFORM

Our SmartCam [Bramberger et al. 2006] provides the hardware and basic software platform for the implementation of our novel software framework. This section summarizes the embedded platform; more details can be found in Bramberger [2005] and Doblander et al. [2006a].

3.1 SmartCam Hardware Platform

Figure 1 depicts the hardware architecture of our SmartCam, which is comprised of a sensing unit, a processing unit, and a communication unit. A CMOS

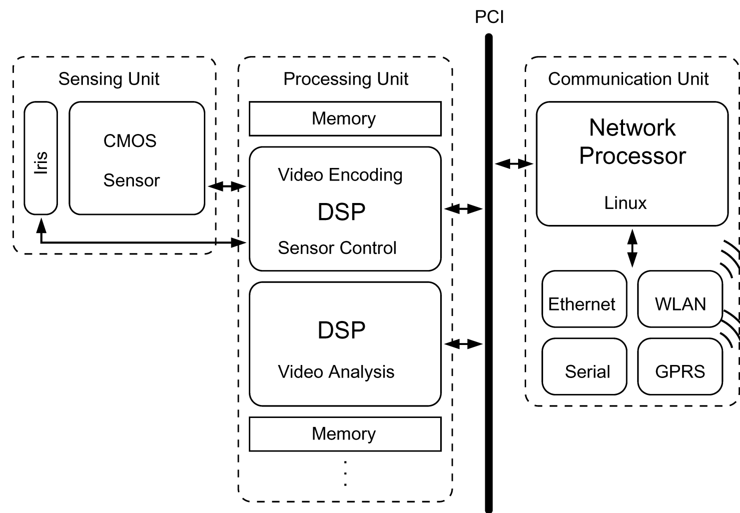

Fig. 1. The SmartCam hardware architecture. It comprises a sensing unit, a processing unit, and a communication unit. Up to 10 DSPs provide the necessary computing power for video analysis algorithms.

image sensor is the heart of the sensing unit. It delivers color images up to VGA resolution at 25 frames per second to the processing unit via a FIFO memory. The processing unit is composed of a variable number of DSPs, which are connected via a local PCI bus. The image processing algorithms are executed on these DSPs. An ARM-based network processor (XScale) controls the communication unit, which has two main tasks. First, it coordinates the internal communication among the DSPs as well as the DSPs and the network processor. Second, it provides IP-based communication channels to the outside world.

3.2 Basic Software Architecture

The software architecture of our smart camera is designed for flexibility and reconfigurability. It consists of several layers, which can be grouped into (i) the DSP-framework (DSP-FW), running on the DSPs and (ii) the SmartCam-framework (SC-FW), running on the network processor. This architecture is based on the abstraction that the application logic is running on the network processor and loads and unloads the actual analysis algorithms onto the DSPs, as needed. An overview of the software architecture of our smart camera is depicted in Figure 2.

SmartCam-Framework. The SC-FW that is illustrated in the left part of Figure 2 serves two main purposes. First, it provides an abstraction of the DSPs to ensure platform independence of the application layer. Second, the application layer uses the provided communication methods (i.e., internal messaging to the DSPs and external IP-based communication) to exchange information or offer data relay services for the DSP-FW. Modules of this part of the software architecture support application development in that they provide high-level interfaces to DSP algorithms and functions of the DSP-FW. Especially, the

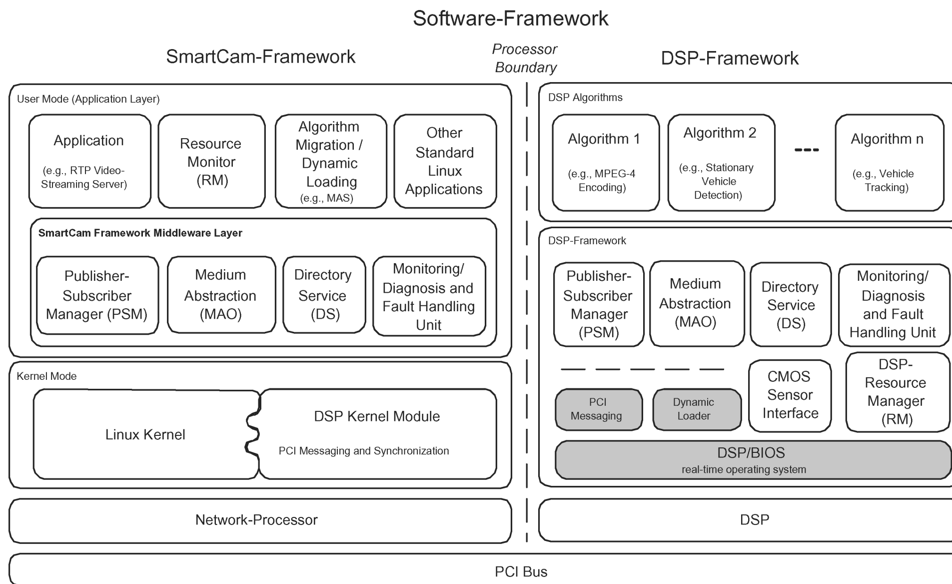

Fig. 2. The overall software architecture of our smart camera. In the left part of the figure, the so-called SmartCam-Framework is illustrated, while the right part shows the so-called DSP-Framework.

mobile agent system makes extensive use of these services to access the DSPs. To further ease application development, the network processor is operated by Linux. Thus, the SC-FW is running on top of a standard Linux kernel.

DSP-Framework. This part of the software architecture, as indicated in the right part of Figure 2, runs on every DSP in the system. The main purposes of the DSP-FW are (i) the abstraction of the hardware and communication channels, (ii) the support for dynamic loading and unloading of application tasks, and (iii) the management of on-chip and off-chip resources of the DSP. Of course, the sensor interface module is only needed on the DSP to which the image sensor is connected. The key functionality in the DSP-Framework is the PS middleware that is described in Section 4.1. These service management facilities are needed to allow algorithms on different DSPs to establish connections to each other, dynamically. The DSP-FW is built upon the DSP/BIOS operating system from Texas Instruments.

Dynamic Loading. All video analysis algorithms and also some framework components can be loaded and unloaded at runtime by the *Dynamic Loader* module. Actually, only modules of the DSP-FW, in dark shade in Figure 2, have to be available at startup. All other components can be dynamically loaded at runtime. Therefore, the framework and the application can easily be extended or adapted to dynamic changes in the system's environment, if desired.

The dynamic loading facilities are also the basis for more sophisticated services like load distribution [Bramberger 2005], dynamic power management [Maier 2006], and graceful degradation to cope with faults.

Table I. Qualitative Comparison of Different Software Architectures

software architecture	*advantages*	*disadvantages*
"no architecture"	efficiency	for single processors only no support for component modeling only static connections / limited scalability
multimedia frameworks	efficiency (ie. streaming) (static) component building limited resource monitoring	limited to specific platform no networking
state machine/ data-flow oriented frameworks	versatile control scheme for fine and coarse grain algorithms	no component modeling only synchronous operation only static configurations
publisher–subscriber architecture	dynamic component composition transparent communication implicit scalability	system-level software required only for coarse grain components
general-purpose middleware	dynamic component composition versatile implicit scalability	heavy resource requirements limited real-time capabilities limited data-streaming programming overhead

4. MIDDLEWARE ARCHITECTURE

Distributed image and video processing are the main applications typically found on smart cameras and visual sensor networks. Our major goal was to develop a middleware that supports the requirements of visual sensor network applications as much as possible:

Flexibility in application composition. The overall application should by easily composed by individual tasks, which follow a transparent communication pattern. The functionality of the individual tasks as well as their provided QoS should be modified dynamically during runtime.

Scalability. The software framework should be scalable concerning the number and type of individual tasks, the available hardware resources, as well as the data volume transferred within the network.

Limited resource consumption. The software framework should carefully utilize the limited resources on the embedded platform (i.e., memory and CPU capacity).

Low-performance overhead. The performance overhead of the software framework should be kept low, since the desired applications demand high-processing, memory and communication capacities.

Support for real-time operation. The runtime behavior of the software framework should be predictable in order to support real-time operation.

We compared the various potential software architectures to select the best fitting architecture for target applications (Table I). Applications using "no-architecture" (only OS-calls) are probably very efficient concerning resource consumption, but are very limited concerning networking, scalability, and flexibility. Multimedia frameworks, such as the MFP from Texas Instruments, provide high-resource efficiency and development support for streaming applications. However, these frameworks are limited to a specific platform and offer

only restricted networking. State machine and data-flow oriented frameworks provide versatile control schemes and can be applied at various levels of abstraction. Their limitations are synchronous communication, static configuration, and limited networking. PS architectures and general-purpose middleware systems provide various networking services, support dynamic component composition, and are scalable with regard to the software components and hardware nodes. They are typically applied for coarse-grained components and require a substantial system-level software on top of the operating system.

Thus, a resource-aware PS architecture would best fit our requirements.

4.1 The Publisher–Subscriber Middleware

The PS architecture is an integral part of the DSP-FW and the SC-FW. It aims at providing seamless and flexible connections between the algorithms running on the DSPs. Furthermore, it has to provide the basic means for supporting application reconfigurations aimed at reducing power consumption or realizing graceful degradation in case of failures.

From the framework's point of view, every video analysis algorithm is a separate entity that is executed in its own thread. Interconnections of the algorithms are defined by the application. In previous work, we used statically defined relations among different data services (i.e., algorithms to simplify intertask communication). This resulted in a very efficient message exchange over the PCI bus. However, the static bindings of data producers and consumers substantially restricted flexibility in dynamically combining algorithms. Furthermore, algorithms had to directly invoke PCI communication primitives, which reduced portability.

To overcome these limitations a publisher–subscriber middleware layer (PS-MW) has been introduced. It provides the algorithms on the DSPs with basic message-oriented communication facilities that are transparent concerning the underlying transport medium. Additionally, a directory service was added to enable dynamic service discovery. It is important to mention that the major goal of our efforts was to provide these services with minimum overhead to save resources on the DSPs.

Each application's algorithm is running in its own task. Communication between algorithms is established via mailboxes, which are available in the DSP operating system. Among the different choices for intertask communication mailboxes provide several advantages, such as (i) transparent and efficient communication and (ii) buffered and unbuffered communication, which allows to realize synchronous as well as asynchronous communication with the same OS mechanism.

In video applications, a large amount of data has to be processed. To use the limited memory of the DSPs efficiently, image data is not copied when sent between algorithms on the same DSP. Only references to actual data are exchanged. Small messages like system commands or monitored performance information are directly posted to mailboxes.

Figure 3 depicts the situation for two algorithms residing on the same DSP. The first algorithm provides a data service X that the second uses for further

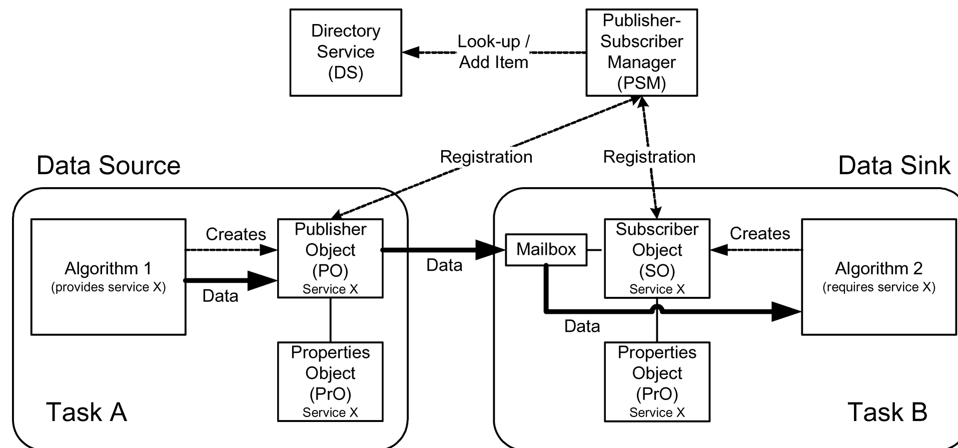

Fig. 3. Fundamental relations between objects of the publisher–subscriber architecture. Only local connections within a single DSP are sketched.

processing. The core of our PS architecture is realized as an efficient object-oriented implementation. A detailed description of the architecture objects is presented in previous papers [Doblander et al. 2006a, 2006b].

4.2 Dynamic Component Composition

4.2.1 *Dynamic Loading and Reconfiguration.* A central aspect of our smart cameras is the dynamic loading and unloading of video analysis algorithms at runtime. The dynamic loader module from Texas Instruments is able to dynamically link and load DSP binaries and has been integrated into the DSP-FW.

Furthermore, each algorithm has to support different QoS levels that can be changed at runtime. A required change in the QoS configuration is signaled by the DSP-FW using a special command message type. Commands are not time-critical and are, therefore, not treated as important as normal data services with tight timing requirements.

In general, there are two different types of trigger sources for reconfiguration actions. One source of triggers for these reconfigurations are alarms generated by the analysis algorithms. Another possibility for triggering a reconfiguration are events raised by internal system-level services like the load distribution service [Bramberger et al. 2005], the power management facility [Maier et al. 2005], or a failure management service.

4.2.2 *DSP Algorithm Component Model.* To support the dynamic reconfiguration of algorithms (i.e, their composition and change of attributes) in our surveillance applications it is necessary for each algorithm to comply with a special component model—the DSP algorithm component model (DACM)—as indicated in Figure 4. The DACM is based on the XDAIS algorithm component model from Texas Instruments [Instruments 2002]. It extends the XDAIS model to support dynamic loading and the PS communication scheme, as well

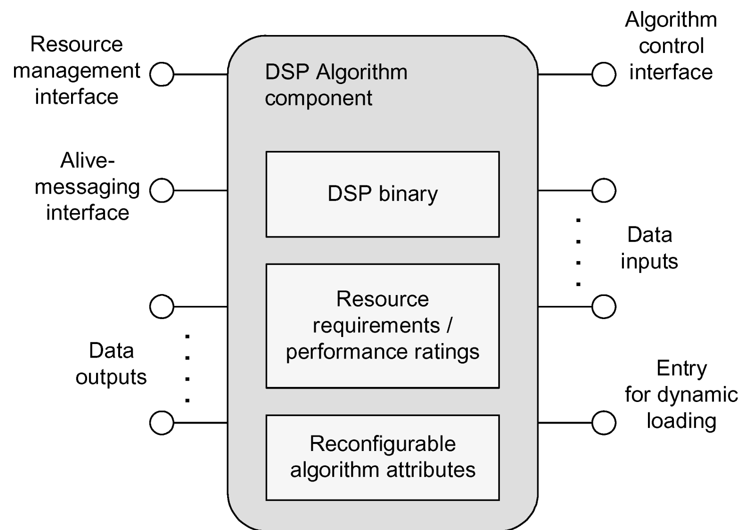

Fig. 4. Principle structure of a DACM component.

as by adding crucial entries in the algorithm's resource descriptions to address all critical system resources. In the XDAIS model, the focus is on design time integration and, therefore, resource ratings are only provided in the component documentation.

In the DACM, all components have to provide all their resource information at runtime to allow for dynamic component composition. The framework further defines the necessary interfaces and algorithm descriptions that are required to load an algorithm at runtime. Only algorithms following the DACM can be dynamically composed at runtime. Algorithm characteristics that have to be exhibited by each algorithm component are collected in Table II.

In the framework, the resource manager module keeps track of already allocated resources and available resources. Based on this information and the algorithm characteristics, the framework can decide whether a component can be (dynamically) integrated into the system. Note that the enhanced direct memory access controller (EDMA) of the DSPs is a critical resource as image analysis is very memory intensive and data is mostly copied by EDMA to keep CPU load as low as possible.

4.2.3 *Resource Monitoring.* The PS-MW has to ensure proper component composition when new algorithms are loaded at runtime. As a basis, the framework uses the component resource descriptions provided by each algorithm following the DACM to determine the component's resource requirements. Now, to decide on the feasibility of a composition, the available resources in the system have to be calculated and compared to the resource requirements. The resource monitoring module in the framework constantly computes the resource loading.

Countable resource metrics like the number of used EDMA channels, EDMA tables, and EDMA transfer complete interrupts are quite easy to determine for each algorithm. In the software framework, this is achieved by a EDMA manager that is the only authority to request EDMA related resources. Therefore,

Table II. Example Algorithm Information as Provided by the DACM

Required Services from other components
QoS levels
Resource requirements
EDMA channels and their priorities
EDMA tables
EDMA interrupts
Performance Ratings
CPU utilization for each QoS level
Transfer frequency of each EDMA channel
Transfer length of each EDMA channel

it is also easy to check whether a component's resource requirements can be met by a simple comparison of available and demanded resources. Only if sufficient resources are available, the component is loaded and started. The actual composition is then simply realized by the PS-MW. All required data services are looked up and connected adequately, as described in Section 4.1.

On the other hand, it is quite hard to provide exact characteristics of more complicated resource metrics like CPU utilization, PCI bus utilization, and EDMA controller utilization—they are also subject to constant fluctuations, which makes accurate a priori characterization impossible. However, these metrics are typically critical in terms of real-time operation of the system. As they are dynamically changing, it is necessary for the framework to observe them constantly. If limits are going to be violated, the framework initiates a graceful degradation in QoS of less important algorithms. That is, the QoS levels of low-priority algorithms are reduced. Prioritization of algorithms is used to control the QoS adaptation. An importance value defines the priority for each algorithm. This value specified by the application developer. The implicit assumption for this procedure is that a lower QoS level results in reduced resource utilization.

Information about PCI bus utilization is not part of an algorithm description. As algorithms are composed at runtime, it cannot be determined a priori by the algorithm designer whether local mailbox communication or remote PCI communication will be used at algorithm deployment. However, for system stability, it is important not to overload the PCI bus. Therefore, PCI utilization is monitored by the resource manager on the network processor. To do so, it collects measurements of the traffic through the medium abstraction objects (MAOs) of all DSPs and the network processor. This is possible because the MAO is the unit on each processor where all traffic to other processors is routed through. Therefore, overall PCI bus load in a single SmartCam i (i.e., $Load_{PCI,i}$) can be computed as

$$Load_{PCI,i} = Load_{PCI,XScale} + \sum_{n=1}^{N} Load_{PCI,DSP_n}, \tag{1}$$

where N is the number of DSPs and $Load_{PCI,XScale}$ and $Load_{PCI,DSP_n}$ denote the load in bytes per second measured at the MAO of the XScale and DSP n, respectively.

Table III. Overview of the Monitored Resources in the PS Middleware

Resource	*monitoring method*	*monitoring period*
CPU utilization (at each QoS-level)	DACM description	at initialization
EDMA (channels, tables, interrupts	DACM description	at initialization
memory limits (static, dynamic)	DACM description	at initialization
PCI load	tracking transfer rates (MAO)	at each invocation
EDMA load	tracking EDMA controller	at each invocation
dynamic memory usage	tracking OS-calls	at each invocation
execution times	measurements by hooks	at message rate
communication times	measurements by hooks	at message rate

Utilization of the EDMA resources on the DSPs is a critical metric for overall system performance because image data is mostly transferred by EDMA. If the EDMA subsystem is overloaded, the timely operation of all algorithms is at risk. To improve the reliability of the system especially with respect to timeliness, it is necessary to avoid resource overloading. EDMA controller load generated from an algorithm is estimated from the algorithm's characteristics provided by the DACM. It can be noted as $Load_{EDMA} = \sum Load_{EDMA,l}$, where $l = 1, \ldots, L$ are the L hardware priority queues of the EDMA controller and

$$Load_{EDMA,l} = \sum_{c=1}^{K} length(c,l)\, freq(c,l) \qquad (2)$$

denotes the transfer bandwidth of priority queue l taking into account all of the K channels c. The function $length(c,l)$ yields the number of bytes transferred on channel c if channel c is assigned priority l. It returns zero for all other values of l. Similarly, $freq(c,l)$ yields the number of transfers issued per second on channel c iff c is assigned priority l.

The third critical system resource is memory. As the PS-MW provides a dynamic environment, it is key to estimate dynamic memory usage of algorithms and to monitor dynamic memory availability. Maximum buffer sizes are known at design time. Therefore, algorithm resource descriptions can be made quite accurate. Monitoring of free dynamic memory resources is done by querying operating system memory management calls.

Table III summarizes the monitored resources in our PS-MW. Note that the first three resource parameters are specified in the DACM description and are checked at the initialization time of the (new) components. The remaining resource parameters are monitored continuously. Execution and communication times are measured at message transfer rates. Thus, for most algorithms, the measurement rate corresponds to the frame rate of the sensor data. The determination of the component performance from these parameters is described in Section 4.2.5.

4.2.4 *Component Composition.* Given the resource requirements information in the algorithm description of the DACM and the continuous monitoring of actual resource occupancy, as described in Section 4.2.3, the basic step of the composition process is a comparison of required and available resources.

The algorithm is loaded by the dynamic loader facility when the algorithm's resource requirements are met. On load of the algorithm it registers with the

PSM (i.e., it queries for services it requires and publishes services it provides). In this respect, our approach is somewhat different to other component-based middleware because the algorithm is loaded even if required services are currently not available in the system. However, then the algorithm is put to sleep, because it cannot do its work. But if at a later time another component is inserted that provides the missing service, then the sleeping algorithm is brought back to work by the PSM. With this simple mechanism, we can load algorithms without worrying about the sequence of algorithms defined by data-flow dependencies.

Another relaxation in our component composition approach compared to standard middleware technology is that there is some degree of freedom concerning service querying. In general, it is necessary that service interfaces (i.e., output of one component and input of another component) completely match in order to be connected. This is in principle also true for this approach, but with the introduction of different QoS levels it is also possible for a component to accept services that do not match up to a certain extent. Of course, it is required that key attributes have to match. But it is up to the algorithm to decide which ones it is able to accept even if diverting.

In that respect, it is possible that there are several services available in the system that potentially match new components requirements. Then this component has to choose one of these. Generally, the one with the highest QoS level would be the best choice. But especially in abnormal situations like failure conditions and the like, the situation might be different. Then, it could be the case that using a lower-quality service can allow the algorithm to at least provide rudimentary functionality. This is a basic feature that is exploited when graceful degradation is used to cope with faults that lead to resource failures.

4.2.5 *Component Performance Monitoring.* It is important for several reasons to continually monitor all components in the system for their performance. First, it allows the framework to reason about likely deadline misses that compromise real-time operation. Second, performance measurements can be used to reason about the fitness of components, which is important for fault-tolerance mechanisms.

4.2.5.1 *Dynamic Memory Usage.* Especially, memory consumption of a component observed over time can exhibit buffer management problems in algorithms or other memory leaks. Of course, only dynamic memory allocation in heap memory is observed. Operating system primitives are used to determine current memory usage for each task in the system. This is sufficient, since every algorithm runs in its own execution task.

4.2.5.2 *Execution Time.* Execution times are constantly measured by hooks in the PS-MW at the inputs and the outputs of all algorithms (Figure 5). That is, a system counter is captured each time a hook function is called in a subscriber or a publisher, respectively. By this mechanism current computation time in CPU cycles is determined as the difference $T_{A_i,exec} = |T_{A_i,out} - T_{A_i,in}|$, where $T_{A_i,in}$ represents the counter value at the time when all inputs of algorithm A_i were ready. $T_{A_i,out}$ stands for the counter value when all outputs of

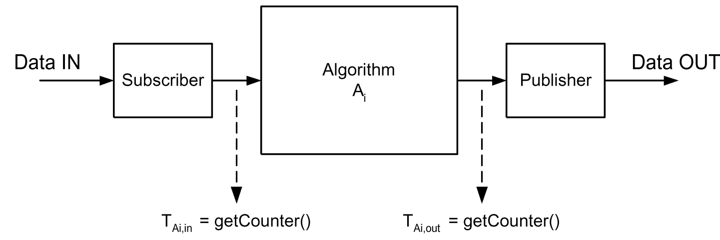

Fig. 5. Basic mechanism to monitor the algorithm execution times.

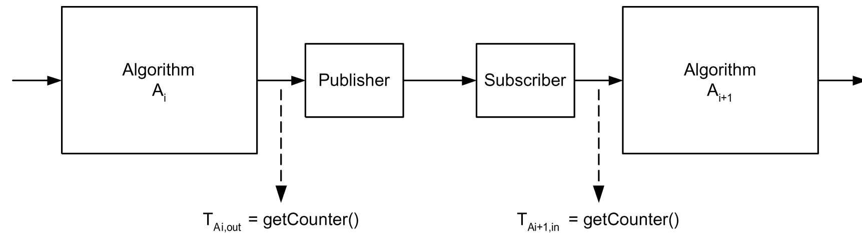

Fig. 6. Basic mechanism to monitor the communication delay from algorithm A_i to algorithm A_{i+1}.

algorithm A_i were ready. Image analysis algorithms are typically implemented such that they take some input data, process it, and produce some output data on a frame-by-frame basis. Therefore, measuring the CPU cycles from the moment an algorithm receives data to the moment it posts the output is a good estimate for its execution time.

As a side product the input frame rate of an algorithm can be checked by observing two subsequent input counter values $T_{A_i,in}[n]$ and $T_{A_i,in}[n+1]$. An estimate $E_{f_{frame}}[n+1]$ for the current input frame rate of algorithm A_i at sample time $n+1$ is then given by

$$E_{f_{frame}}[n+1] = \frac{f_{CPU}}{|T_{A_i,in}[n] - T_{A_i,in}[n+1]|} \tag{3}$$

where f_{CPU} denotes the clock frequency of the CPU. By continually observing these frame rate estimates, problems can be detected early so that interventions are likely to prevent failures.

4.2.5.3 *Communication Delay*. Another performance rating that can be observed by the framework is communication delay. That is, the delay from a publisher to its associated subscribers is evaluated. As the execution time, the communication delay is also an estimate based on capturing a counter at well-defined interaction points in the PS subsystem.

In Figure 6, the principle is illustrated for two algorithms A_i and A_{i+1}, respectively. Note that the same measurement points are involved as used for the execution time estimation. But in this case, the probe points of different algorithms are used.

The estimate $E_{A_i \rightarrow A_{i+1}}[n]$ for the communication delay at sample time n can be written as

$$E_{A_i \rightarrow A_{i+1}}[n] = \frac{|T_{A_i,out}[n] - T_{A_{i+1},in}[n]|}{f_{CPU}} \tag{4}$$

with f_{CPU} being the CPU clock frequency.

Observing communication delays over a certain period can reveal timing problems. Possible causes could be high loads on the CPU or the PCI bus. Single absolute values of communication delay can be used to uncover real-time problems. For example, the sum of all execution times and communication delays in a processing chain determine the maximum possible frame rate at the system perspective. The overhead introduced by hooks is neglectable since the hooks require only simple table look-ups and are typically called at rates of tens or hundreds of milliseconds.

5. MIDDLEWARE-BASED FAULT TOLERANCE

Monitoring the various performance parameters over time provides valuable information about the overall status of the application. By applying adequate reactions when certain resource limits are reached, we can increase the availability of the application. We incorporate this approach into our PS-MW to take a step toward autonomous operation of smart cameras. Integrating simple fault-tolerance mechanisms into our middleware helps to reduce the application development time as well as to reduce the overall code size since the fault-tolerance code has to be included only once in the whole system. The main management parts of this fault-tolerance architecture (FTA) are hosted on the network processor within the SC-FW, whereas mainly low-level monitoring is included in the DSP-FW of each DSP and simple fault-handling mechanisms are available.

5.1 Fault Handling in a Network of Smart Cameras

The principle idea of our FTA is to introduce some degree of fault tolerance by embedding simple mechanisms in our middleware. We avoid additional hardware components or sophisticated software replication techniques to keep our middleware light-weight. However, it is possible to exploit domain-specific knowledge in our DSCs to provide some fault tolerance. Simple methods are used to detect and localize faults and then graceful degradation is employed to mitigate fault effects to prevent system failures.

In order to achieve fault tolerance, we exploit two basic mechanisms of our software framework: dynamic reconfiguration and QoS adaptation. By migrating software components from a faulty to a healthy processor during runtime, several services may become operational again—potentially on a different camera. QoS adaptation may free resources on individual nodes, since lower QoS-levels typically demand for lower memory and computing resources. Thus, in situations with reduced available resources (either due to a hardware failure or a corrupt software module), we may still provide some services.

Table IV. Considered Fault Classes, Detection Methods and Fault-Handling Procedures

fault classes	*detection mechanisms*	*fault handling procedures*
algorithm faults	plausibility checks resource monitoring	algorithm reload graceful degradation
communication faults	alive messages	node reboot
hardware faults	watchdog alive messages	DSP reboot node reboot

5.1.1 *Considered Fault Classes.*

In our work, we currently consider only a subset of all possible faults within a single SmartCam or in a network of collaborating SmartCams.

Algorithm faults. The main focus of the FTA is to provide some higher-level fault detection for algorithms based on application-specific knowledge. That is, analysis results of different algorithms are often related to each other. For example, in case of a traffic jam two stationary vehicle detection algorithms on two adjacent cameras have to come to the same decision—at least after some limited time interval. If an algorithm detects the traffic jam and the other fails to do so it can be deduced that one of them exhibits incorrect behavior. There are other cases where inconsistent observations of two or more algorithms suggest a failure of one of them. Exceeding the specified resource limits might be another reason for an algorithm fault.

Communication faults. It is essential for a network of cooperating smart cameras to have mutual communication paths readily available to exchange information about the observed scene. The middleware framework employs a simple messaging protocol between neighboring nodes to check mutual reachability. These so called alive- or heartbeat-messages are exchanged regularly. Missing messages over a preset time span results in the corresponding node to be considered as down.

Hardware faults. As the main focus of the FTA is handling software problems, it makes sense to treat several hardware problems as well. This fault class is considered mainly because it is relatively easy to handle them in the given framework for dynamic reconfiguration and coping with algorithm problems. In our FTA, we distinguish between a faulty processor (DSP breakdown) and complete breakdown of a SmartCam.

5.1.2 *Fault Handling Procedures.*

To cope with the previously-mentioned fault classes, different counter measures are used by the FTA (Table IV).

Algorithm reload. If an algorithm shows unexpected behavior, it has to be restarted. That is, the algorithm is reloaded on the DSP. This procedure takes only a couple of milliseconds.

As the reboot of a DSP takes significantly longer than the reload of an algorithm, it is advisable to first try if reloading the algorithm in question solves the problem. But if restarting or even repeated restarting does not lead to a successful recovery, rebooting the DSP can be of assistance. Incorrect behavior

of all algorithms running on one DSP indicates a malfunction caused by the DSP and a reboot is necessary.

Graceful degradation. A key mechanism for increasing service availability is to degrade some functionality, despite risking overall system failure. There are two basic means for degrading a service. First, its QoS level can be reduced. A simple but realistic assumption we make here is that higher QoS results in increased resource usage. Second, an algorithm can be shut down completely. Graceful degradation is controlled by an importance parameter, which is assigned to each algorithm. The FTA reduces the QoS level of algorithms with lower-importance parameter first.

DSP reboot. The reboot of a DSP is necessary if a DSP crashed or is under the strong suspicion to have at least partially crashed (e.g., temporary malfunction of the RAM).

Node reboot. In case a node recognizes that it is isolated from the rest of the network, it can decide to undergo a reboot procedure to eliminate possible transient network (stack) problems. Rebooting in this case is unproblematic because if network communication failed, it does not contribute to system goals anymore. But chances are that a transient problem is eliminated after reboot.

Operator notification. An operator (i.e., some global monitoring authority) has to be informed if any unexpected behavior is noticed. If a node repeatedly shows abnormal behavior despite automatic recovery actions, human inspection and maintenance actions are inevitable. Therefore, all detected fault events subsequent counter measures are logged so that an operator can retrieve the information on demand.

Furthermore, a node must always be informed about the operational reliability of its neighbors. That is, if a node diagnoses itself as (partially) faulty, its direct neighbors have to be informed about these fault condition. This is to simplify credibility checks in neighboring nodes. Because if a node testifies itself as faulty, the others can skip the voting process and exclude the faulty camera's results from further consideration.

5.2 Middleware-Based Fault-Tolerance Architecture for Smart Cameras

The SmartCam software framework incorporates the FTA that provides fault tolerance as middleware services. The FTA comprises several units on the network processor and the DSPs. Figure 7 illustrates the principle relationships of the different components.

Every algorithm on the DSP that is subject to the FTA monitoring is registered with the PSM so its input and output connections are known. Furthermore, the algorithm descriptions, as described in Section 4.2.2, provide the basis for decisions on the algorithm's resource usage. Most relevant are the algorithm's name for identification, its current QoS level and what other QoS levels are offered, the resources requirements for the current QoS level, the current importance measure assigned by the application developer, and information about the algorithm's typical execution time. The individual components of our

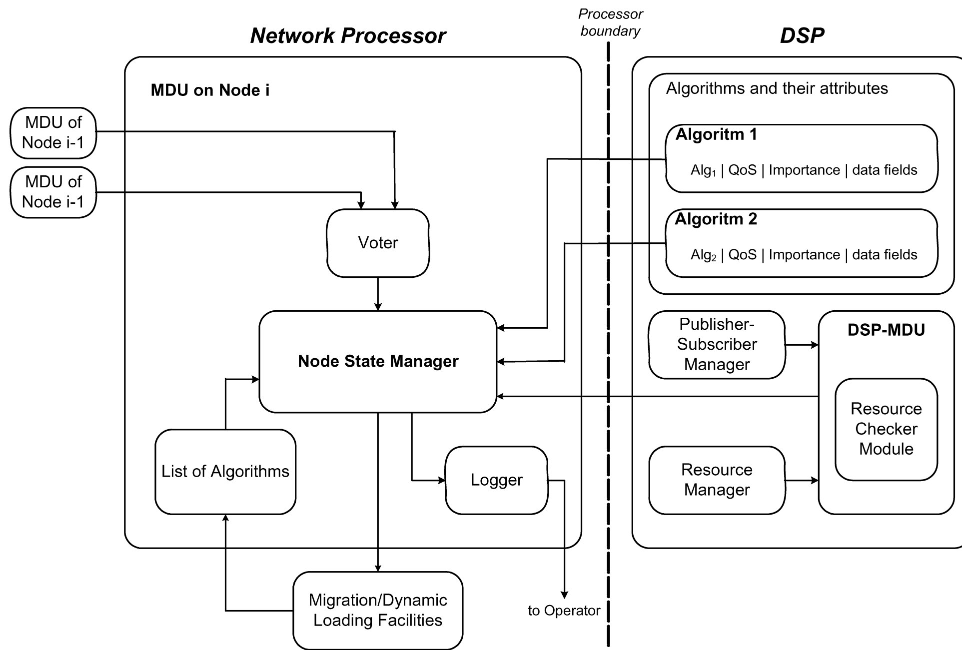

Fig. 7. Overview of the fault-tolerance architecture as it is included in the SmartCam software framework.

FTA can be described as follows whereas the main part of the FTA resides in the so-called monitoring and diagnosis unit (MDU) on the network processor.

Resource Checker Module. The resource checker module (RCM) is the only module in addition to already described framework components of the DSP-FW. The RCM monitors all relevant data concerning projected and actual resource usage of each active algorithm. Resources demanded by the algorithms are compared to the resources available in the system. Necessary resource information is queried from the RM residing on the DSP. The RCM determines whether sufficient resources are available and also communicates its data to the node state manager (NSM) on the network processor.

Node State Manager. The NSM is the central entity of the FTA. It determines a node's state by evaluating data from the RCM, the voter, and the analysis results of the currently active algorithms.

We consider the following states of the node: (i) the *normal state*, where all algorithms work correctly; (ii) the *low-resource state*, where insufficient resources are available; (iii) the *DSP crash state*, where at least one DSP has crashed on the SmartCam; (iv) the *algorithm crash state*, where a malfunctioning algorithm has been detected; and (v) the *communication error state*, where communication to other SmartCams can currently not be established. The transitions among the node's states are controlled by a simple finite state machine.

Most conditions for entering a specific state can be evaluated by our fault-detection mechanisms or by checking the current resource usage. Detecting an

algorithm fault is based on application-specific plausibility checks. These checks often require information from neighboring cameras.[1] The *voter* unit compares the node's analysis results with the results of the two closest neighboring nodes. Deviations are then fed to the NSM that changes the node's state if applicable. A reasonable frequency for voting analysis results from algorithms is once in a few seconds depending on the application requirements.

Logger. The node's state is recorded by the *logger* framework unit. This data can be used to detect abnormal behavior in the long-term behavior of a node such as periodical failures of the hardware or software due to, for example, environmental conditions. Depending on the node's state, appropriate actions are induced by the NSM and recorded by the logger. Furthermore, logged data can be retrieved by remote clients (i.e., operator workstations).

Reloading. In case of the necessity of a reload or unload, the NSM instructs the migration and dynamic loading facility (MDL) to reload or unload, the suspicious algorithm. The MDL induces the reload or, in case of an unload, performs the unload and updates the list of current algorithms residing on the DSP. This list holds information including which algorithm runs on which DSP on this node as well as on the two closest neighboring nodes. In that way, the status quo can be restored after rebooting from a DSP crash.

6. EXPERIMENTAL RESULTS

6.1 Performance Analysis of the Publisher–Subscriber Middleware

The PS middleware has been implemented on our SmartCam prototype, which consists of an Intel IXP425 XScale network processor running at 533 MHz and two Texas Instruments TMS320C6415 DSPs running at 600MHz. The most important performance parameters are presented in the following sections; more implementation details can be found in Doblander et al. [2006b, 2006a].

6.1.1 *Memory Requirements.* An important requirement for the task communication framework on the DSPs of the SmartCam is to use only little memory to save it for the analysis algorithms. Although our middleware has been implemented in C++, the memory footprint is only 15.78KB. The runtime memory consumption is also low (i.e., in the order of several hundred bytes per management object in the framework).

The total memory consumption overhead depends on the number of published services and subscriptions in the system. In a typical setting, there are two algorithms per DSP and each algorithm provides one service and subscribes to one service. Together with the management objects this yields a typical total memory overhead of the middleware of 3.71KB per DSP (Table V).

[1]Consider a simple traffic monitoring scenario where information from neighboring cameras may be exploited to identify an algorithm fault: Three consecutive SmartCams on a highway section report the traffic statistics where the first and the third camera report a traffic jam and the second camera reports high-traffic throughput.

Table V. Memory Requirements and Initialization Times of Management Objects in the PS-MW

Component	Memory [Bytes]	Initialization time [μs]
Publisher–Subscriber Manager (PSM)	472	4.68
Directory Service (DS)	256	9.90
Publisher Object (PO)	192	10.17
Subscriber Object (SO)	96	11.01
Properties Object (PrO)	34-72	NA

Table VI. Message Transfer Times for Plain Mailbox Communication and for a Transfer Using our Publisher–Subscriber Middleware

Transfer Mode	Value [μs]
Mailbox only	1.04
With PS-MW	1.21

6.1.2 *Initialization and Communication Overhead.* As the PS-MW adds some management overhead, we measured the times spent in the initialization phase of the PS-MW at system start-up. This one-time initialization introduced an overhead of only up to about 10 μs (Table V).

To assess the overhead in message transfer time when employing our lightweight PS-MW, we have performed some simple experiments. Several different scenarios have been examined. First, the time spent for a plain mailbox communication between two tasks was measured. After that, the same tasks have been adapted to use the PS-MW. That is, they communicated via a PO at the sender and a SO (including a mailbox) at the receiving task. In this experiment, the time spent from sending the message at the publisher until it had been received at the subscriber was measured.

Note that in this scenario, one publisher with exactly one connected subscriber was examined (i.e., a unicast communication scheme). The results are summarized in Table VI. The overhead in this simple configuration amounts to 16% compared to simple mailbox transfers. This overhead is introduced by the additional setup mechanism of the PS framework. It is independent on the size of the transferred data. However, this overhead is neglectable when large amounts of data, such as an image frame, are transferred and processed.

We also examined the multicast communication scheme (i.e., one publisher with several subscribers connected to it). The significant time measure in this case is the overall time needed to transfer the published message to all subscribed tasks. Again, only tasks on the same DSP were considered. Transfer time increases almost linearly with the number of subscribers.

In another experiment, the transfer times between tasks on different DSPs have been analyzed. The results are summarized in Table VII. The overhead in this case stems from the indirection in the involved proxy mechanism to bridge the PCI bus. It can be seen from the table that multiple subscribers on the same remote DSP yield in a reduced overhead than if they all reside on

Table VII. Message Transfer Overhead Time for Publisher and Subscribers Residing on Different DSPs. Overhead is Given Compared to Direct PCI Transfers Without the PS-MW

Number of SOs	Transfer overhead [μs]		
	2 DSPs	3 DSPs	4 DSPs
1	3.49	—	—
2	4.69	5.24	—
3	5.91	6.44	7.49

different DSPs. This is due to less management overhead in the target. Also note that data is transferred only once to each DSP even if there are multiple subscribers for that data on the DSP.

6.2 Evaluation of the Fault-Tolerance Architecture

The performance of the FTA was evaluated by experiments with dedicated test algorithms (TAs) instead of real video analysis algorithms. A TA is a piece of code suited for the framework that mimics the behavior of a surveillance algorithm. Its output is simulated by a parameterizable data generator. The TA is well suited for fault injection experiments where faulty behavior of video analysis algorithms can be simulated. No real faults have to be provoked and adequate test patterns can be applied. Thus, different fault scenarios can be examined easily.

Every TA is launched as a single task. It implements the standard interfaces needed to be deployed within the software framework. This includes also an appropriate algorithm description as it is required by the framework. The outputs of the TAs are, therefore, as meaningful as those of the actual algorithms with respect for their use with the FTA.

To evaluate the fault-tolerance architecture two key metrics are used as the evaluation criteria:

—the time elapsed to detect a fault and

—the time required for the execution of counter measures.

In order to demonstrate the FTA's ability to detect faults and to illustrate its reactions, two example fault scenarios are presented in the following:

(1) Scenario 1: Inconsistent observations of algorithms on different nodes, and

(2) Scenario 2: A crashed DSP.

The surveillance setting assumed for the two scenarios comprises three smart camera nodes N_{i-1}, N_i, and N_{i+1} along a highway where the cameras are equipped as the prototype described in Section 3.1. It is the assumed application design that three algorithms run on each camera to observe the scene. The algorithms and their attributes of this example application are listed in Table VIII.

The following considerations are based on performance numbers presented in earlier work [Bramberger et al. 2004]. Over the PCI bus, a transfer rate of

Table VIII. Algorithms and Their Attributes for the Example Traffic Surveillance Application

Algorithm	QoS Levels	Importance	Minimum QoS Level
MPEG-4 Encoder (MPEG)	Q_1, Q_2, Q_3	5	Q_2
Stationary Vehicle Detection (SVD)	Q_1, Q_2	2	Q_2
Traffic Statistics (STAT)	Q_1, Q_2	1	—

15 MB/s for communication between a DSP and the XScale is assumed as the lower bound for small messages. The size of the transferred messages is always 128 bits consisting of a 32-bit message ID and 96 bits of data. These include the algorithm's reference number and analysis results (e.g., the considered time interval and the number of vehicles counted during this interval). Therefore, every message sent over the PCI bus via the PS-MW was measured to result in an average transfer time of $t_{msg,PCI} = 0.031$ ms.

6.3 Scenario 1: Inconsistent Observations

Given are three camera nodes N_{i-1}, N_i, and N_{i+1} along a highway. Node N_i is observing an area characterized by stop-and-go traffic and it is in normal mode. It hosts an MPEG-4 encoder (MPEG) on one DSP and a stationary vehicle detection (SVD) on the second DSP. The SVD on node N_i is faulty and, therefore, does not detect any stationary vehicles during time interval t. The two neighboring nodes N_{i-1} and N_{i+1}, however, register a number of v_{i-1} and v_{i+1} stationary vehicles during time interval t, respectively.

6.3.1 *System Response to Scenario 1.* As node N_i's neighbor's observations are not consistent with those of node N_i, a plausibility check will eventually detect this inconsistency (i.e., the voter's output does not match the SVD's output and the NSM indicates a malfunction of the SVD). The NSM instructs the MDL to reload the SVD. It sets the node to inconsistent observation mode and sends this information to the logger. The SVD is then reloaded and initialized. As most problems with algorithm's detection results are due to transient buffer problems, it is likely that the reinitialization solves the problem and the algorithm works properly again. If the problem persists, the algorithm has to be removed and the operator has to be notified.

6.3.1.1 *Time to Fault Detection.* The SVD sends its output every 2s to the ACM and the voter, respectively. The voter's output is sent to the NSM, thus, two messages have to be sent and the first message is sent τ seconds after the occurrence of the fault. The voter's output reaches the NSM within a time interval of

$$t_{detection} = \tau + 2 \cdot t_{msg,PCI} = \tau + 0.062\text{ms} \tag{5}$$

where $\tau \leq 2$s is the interval of alive messages specified in the framework.

6.3.1.2 *Time Required for Counter Measures.* To handle the problem the FTA sends one or more messages to the MDL to reload the SVD. The time $t_{reload,SVD}$ required for reloading the SVD was measured to be 46 ms. Additional

Table IX. Resource Requirements for Surveillance Tasks According to Bramberger et al. [2004]

Algorithm	QoS Level	QoS Description	CPU [MIPS]	RAM internal	RAM external
MPEG-4	Q_1	PAL 20 fps	2,840	400 kB	0
MPEG-4	Q_2	PAL 10 fps	1,920	400 kB	0
SVD	Q_1	CIF 12 fps	3,600	500 kB	17 MB
SVD	Q_2	QCIF 12 fps	900	330 kB	4 MB

time $t_{reg,SVD} = 1$ ms for registering and starting the algorithm, has also to be considered. Initialization of the SVD takes approximately 10 frames, which corresponds to $t_{init,SVD} = 500$ ms in case of QoS level Q_2 with 10 fps. The total time spent on counter measures adds up to

$$t_{counter} = t_{msg,PCI} + t_{reload,SVD} + t_{reg,SVD} + t_{init,SVD} \quad (6)$$

$$t_{counter} = 0.031\text{ms} + 46\text{ms} + 17\text{ms} + 500\text{ms} \quad (7)$$

$$t_{counter} = 563.031\text{ms} \quad (8)$$

It can be seen from this result that the overhead of the FTA is negligible compared to the algorithm-specific reinitialization times. Of course, algorithms with less initialization time result in less out time of the service in case of necessary reconfiguration.

6.4 Scenario 2: DSP Crash

In this scenario, the MPEG encoder operates at QoS level Q_1 and an importance of 5 on DSP 1. Additionally, the SVD runs with QoS level Q_1 and an importance value of 2 on DSP 2. Then, the DSP 2 crashes and cannot be rebooted so that the system has to proceed with only one remaining DSP.

6.4.1 *System Response to Scenario 2.* As the network processor maintains a list of currently active algorithms along with their importance values and DSP assignments, the node state manager (NSM) can determine that the SVD algorithm is missing on the node.

This is because DSP 2 has not reacted to polling from the MDU for 1s. A message is sent to the NSM informing that DSP 2 crashed. As resources demanded by the MPEG encoder and the SVD on the highest QoS level exceed the remaining DSP's computational power of 4,800 MIPS (Table IX), the NSM has to reconfigure the system. Note that image scaling and the shutter control for the image sensor takes approximately 1,900 MIPS, which is also considered by the NSM.

Since the SVD has the lower importance the NSM calculates whether it is possible to have the MPEG encoder run on QoS level Q_1 and the SVD on QoS level Q_2. Hence this is not feasible due to the previously-mentioned overhead of 1,900 MIPS, the NSM subsequently determines that it is possible to run the MPEG encoder on Q_2 in combination with the SVD on Q_2. In that way, the node chooses to gracefully degrade the QoS as opposed to a degradation of the service availability. The NSM instructs the MDL to load the SVD onto DSP 1, the MPEG encoder's QoS level is adjusted, and relevant information is sent to the logger. When loading of the SVD onto DSP 1 is finished, the procedure is complete.

6.4.1.1 *Time to Fault Recognition.* The time elapsed until the fault is recognized is primarily determined by the polling interval $t_{polling}$. By adding the transfer time of the notification message $t_{msg,PCI}$ from the polling interface results in a detection time $t_{detection}$ of

$$t_{detection} = t_{polling} + t_{msg,PCI} \tag{9}$$
$$t_{detection} = 1000\text{ms} + 0.031\text{ms} \tag{10}$$
$$t_{detection} = 1000.031\text{ms}. \tag{11}$$

6.4.1.2 *Time Required for Counter Measures.* Again the measures for $t_{reload,SVD}$, $t_{reg,SVD}$, and $t_{init,SVD}$ of the SVD introduced in Section 6.3.1 can be used to compute the time for handling the problem. Additionally, the times for readjusting the MPEG-4 encoder's QoS level and the time $t_{adapt,MPEG}$ for the encoder adapting to the new QoS level have to be considered. The MPEG needs only one frame for adaptation, which corresponds to $t_{adapt,MPEG} = 100$ ms, respectively. Furthermore, two message sending times are involved in the handling of this scenario. First, the MDL has to be notified to reload the SVD. Second, the MPEG encoder has to be commanded to switch to QoS level Q_2. Therefore, the time for necessary reconfigurations to handle the detected problem computes to

$$t_{counter} = 2 \cdot t_{msg,PCI} + t_{reload,SVD} + t_{reg,SVD} + t_{init,SVD} + t_{adapt,MPEG} \tag{12}$$
$$t_{counter} = 0.062\text{ms} + 46\text{ms} + 17\text{ms} + 500\text{ms} + 100\text{ms} \tag{13}$$
$$t_{counter} = 663.062\text{ms}. \tag{14}$$

6.5 Summary

Both of the described scenarios show that the detection and reconfiguration overhead is dominated by algorithm-specific initialization times. Current algorithm implementations often rely on building some kind of models of the scene. The quality of the analysis depends strictly on the quality of the models. Therefore, many frames are used to build up the models before actual analysis is performed. The encoder algorithms are better in this respect as they do not rely on sophisticated scene models.

The presented brief results are based on quite restrictive figures for communication times. That is, typically communication is much faster over the PCI bus. But to have some upper limit of the detection times the minimum measured PCI speed was considered.

7. CONCLUSION

In this article, we have presented a novel middleware for embedded smart camera networks. This middleware is based on a very resource-aware PS architecture that supports synchronous and asynchronous communication between tasks in the given dynamic application environment. Our middleware supports dynamic component composition and enables dynamic task reconfiguration during runtime—both of which are quite unusual in such resource-limited distributed embedded systems.

Although the middleware has been implemented on a SmartCam network and evaluated in a traffic monitoring application, this research might be useful

for a broader research community. The major lessons we have learned throughout this research can be summarized as follows:

—There is currently a strong trend towards *visual sensor networks*, which process visual data directly at the sensor nodes and stream abstracted data throughout the network. The recent developments in smart camera networks [Aghajan and Kleihorst 2007; Rinner and Wolf 2008b; Rinner et al. 2008] demonstrate this trend very well. As briefly discussed in the introduction, visual sensor networks have different requirements on hardware and software compared to traditional sensor networks. Our middleware addresses these requirements and is, therefore, applicable to various visual sensor networks as well.
—Monitoring relevant and critical resources is crucial in distributed embedded systems. This is especially important in dynamic software environments including methods for dynamic loading and reconfiguration. Most middleware systems, however, monitor only "standard" resources such as CPU utilization and global memory consumption. Thus, much care must be taken for identifying and monitoring critical resources such as DMA, communication loads, and the memory consumption for individual segments.
—Our middleware provides mechanisms for dynamic reconfiguration, QoS adaptation, and resource monitoring. By exploiting these mechanisms, we have integrated simple but effective fault-tolerance methods in our middleware. Due to its intented simplicity, only very limited guaranties concerning the fault-tolerance behavior can be achieved. However, our experiments show that in "real" applications the availability can be significantly improved. Note that our fault-tolerance architecture is transparent to the application developer and causes virtually no additional overhead.

There are several directions for future work. A natural direction is to further explore the fault tolerance mechanisms of the middleware framework with the goal of increasing the overall system's availability. Another line of research would be to include additional host and networking services in the middleware framework (compare [Rinner et al. 2007]). These services strongly support the development of distributed applications. A more general research approach would deal with the design process of distributed applications based on middleware frameworks. Important topics for this direction are composability, scalability, and portability of the distributed application. Finally, the middleware framework will be demonstrated in different application scenarios.

ACKNOWLEDGMENTS

This work has taken place at the Institute for Technical Informatics, Graz University of Technology.

REFERENCES

Aghajan, H. and Kleihorst, R., Eds. 2007. *Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC'07)*. ACM, New York.

AKYILDIZ, I. F., SU, W., SANKARASUBRAMANIAM, Y., AND CAYIRCI, E. 2002. Wireless sensor networks: a survey. *Comput. Netw. 38*, 4, 393–422.

BALASUBRAMANIAN, K., WANG, N., GILL, C., AND SCHMIDT, D. C. 2003. Towards composable distributed real-time and embedded software. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, Los Almitos, CA, 226–233.

BECKER, C., SCHIELE, G., GUBBLES, H., AND ROTHERMEL, K. 2003. BASE—a micro-brokerbased middleware for pervasive computing. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*. IEEE, Los Alamitos, CA, 443–451.

BRAMBERGER, M. 2005. Distributed dynamic task allocation in clusters of embedded smart cameras. Ph.D. thesis, Institute for Technical Informatics, Graz University of Technology, Graz, Austria.

BRAMBERGER, M., BRUNNER, J., RINNER, B., AND SCHWABACH, H. 2004. Real-Time video analysis on an embedded smart camera for traffic surveillance. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, Los Alamitos, CA, 174–181.

BRAMBERGER, M., DOBLANDER, A., MAIER, A., RINNER, B., AND SCHWABACH, H. 2006. Distributed smart cameras for surveillance applications. *Computer 39*, 2, 68–75.

BRAMBERGER, M., RINNER, B., AND SCHWABACH, H. 2004. An embedded smart Camera on a scalable heterogeneous multi-DSP system. In *Proceedings of the European DSP Education and Research Symposium*.

BRAMBERGER, M., RINNER, B., AND SCHWABACH, H. 2005. A method for dynamic allocation of tasks in clusters of embedded smart cameras. In *Proceedings of the International Conference on Systems, Man and Cybernetics*. IEEE, Los Alamitos, CA, 2595–2600.

CLARKE, M., BLAIR, G. S., COULSON, G., AND PARLAVANTZAS, N. 2001. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, R. Guerraoui, Ed. Lecture Notes in Computer Science. Vol. 2218. Springer, 160–178.

DEMICHIEL, L. G. 1995. The component object model specification. Tech. rep., Microsoft Corporation.

DEMICHIEL, L. G. 2002. Enterprise JavaBeans specification version 2.1. Tech. rep., SUN Microsystems.

DOBLANDER, A., RINNER, B., TRENKWALDER, N., AND ZOUFAL, A. 2006a. A light-weight publisher-subscriber middleware for dynamic reconfiguration in networks of embedded smart cameras. In *Proceedings of the 5th World Scientific and Engineering Academy and Society International Conference on Software Engineering, Parallel and Distributed Systems*. ACM, New York.

DOBLANDER, A., RINNER, B., TRENKWALDER, N., AND ZOUFAL, A. 2006b. A middleware framework for dynamic reconfiguration and component composition in embedded smart cmeras. *WSEAS Trans. Comput. 5*, 3, 574–581.

FRAGA, J., SIQUEIRA, F., AND FAVARIM, F. 2003. An adaptive fault-tolerant component model. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, Los Alamitos, CA, 179–186.

HANSSON, H., ÅKERHOLM, M., CRNKOVIC, I., AND TÖRNGREN, M. 2004. SaveCCM—a component model for safety-critical real-time systems. In *Proceedings of the 30th EUROMICRO Conference*. IEEE, Los Alamitos, CA, 627–635.

KARSAI, G., SZTIPANOVITS, J., LEDECZI, A., AND BAPTY, T. 2003. Model-integrated development of embedded software. *Proc. IEEE 91*, 1, 145–164.

LIN, C. H., WOLF, W., DIXON, A., KOUTSOUKOS, X., AND SZTIPANOVITS, J. 2006. Design and implementation of ubiquitous smart cameras. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*. IEEE, Los Alamitos, CA, 32–39.

MAIER, A. 2006. Dynamic power-aware camera configuration in distributed embedded surveillance clusters. Ph.D. thesis, Institute for Technical Informatics, Graz University of Technology, Graz, Austria.

MAIER, A., RINNER, B., AND SCHWABACH, H. 2005. A hierarchical approach for energy-aware distributed embedded intelligent video surveillance. In *Proceedings of the IEEE/IFIP International Workshop on Parallel and Distributed Embedded Systems*. IEEE, Los Alamitos, CA, 12–16.

MASCOLO, C., CAPRA, L., AND EMMERICH, W. 2002. Mobile computing middleware. In *Advanced Lectures on Networking: NETWORKING 2002 Tutorials*, E. Gregori, G. Anastasi, and S. Basagni, Eds. Lecture Notes in Computer Science, vol. 2497. Springer, Berlin, Germany, 20–52.

MICROSOFT. 2005. .Net Home Page. http://www.microsoft.com/net.

MODY, M. 2006. XDAIS-DM (XDM): A step towards the "plug and play" architecture for multimedia codecs. TI Developer Conference. http://www.s.ti.com/sc/techlit/sprp496.pdf.

MOLLA, M. M. AND AHAMED, S. I. 2006. A survey of middleware for sensor Networks and Challenges. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW'06)*. IEEE, Los Alamitos, CA, 223–228.

OBJECT MANAGEMENT GROUP. 2001. Real-Time CORBA 2.0. http://www.omg.org.

OBJECT MANAGEMENT GROUP. 2002. Minimum CORBA 1.0. http://www.omg.org.

OBJECT MANAGEMENT GROUP. 2005. http://www.omg.org/technology/documents/formal/components.htm.

PITT, E. AND MCNIFF, K. 2001. *Java.rmi: The Remote Method Invocation Guide*. Addison Wesley, Upper Saddle River, NJ.

POPE, A. 1998. *The CORBA Reference Guide: Understanding the Common Oject Request Broker Architecture*. Addison Wesley, Upper Saddle River, NJ.

RINNER, B., JOVANOVIC, M., AND QUARITSCH, M. 2007. Embedded middleware on distributed smart cameras. In *Proceedings of the IEEE International Conference on Acoustics, Speech, Signal Processing (ICASSP'07)*. IEEE, Los Alamitos, CA, 1381–1384.

RINNER, B., SCHRIEBL, W., WINKLER, T., QUARITSCH, M., AND WOLF, W. 2008. The evolution from single to pervasive smart sameras. In *Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC'08)*. ACM, New York.

RINNER, B. AND WOLF, W. 2008a. A bright future for distributed smart cameras (guest editor's introduction). *Proc. IEEE 96*, 10, 1562–1564.

RINNER, B. AND WOLF, W. 2008b. An introduction to distributed smart cameras. *Proc. IEEE 96*, 10, 1565–1575.

SCHMIDT, D. C. 2002. Middleware for real-time and embedded systems. *Comm. ACM 45*, 6, 43–48.

SESSIONS, R. 1997. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, NY.

SYSTEMS, M. C. AND THALES. 2003. Light Weight CORBA Component Model. Tech. rep., Object Management Group.

TEXAS INSTRUMENTS. 2002. TMS320 Algorithm Standard—Rules and Guidelines. Literature Number: SPRU352E.

WOLF, W., OZER, B., AND LV, T. 2002. Smart cameras as embedded systems. *Computer 35*, 9, 48–53.

Received May 2007; revised January 2008; accepted September 2008