Bernhard Rinner [*] Markus Quaritsch

Klagenfurt University, Pervasive Computing Group
Lakeside B02
9020 Klagenfurt, Austria

Embedded Middleware for Smart Camera Networks and Sensor Fusion

Abstract

Smart cameras are an interesting research field that has evolved over the last decade. In this chapter we focus on the integration of multiple, potentially heterogeneous, smart cameras into a distributed system for computer vision and sensor fusion. An important aspect for every distributed system is the system-level software, also called middleware. Hence, we discuss the requirements on middleware for distributed smart cameras and the services such a middleware has to provide. In our opinion a middleware following the agent-oriented paradigm allows to build flexible and self-organizing applications that encourage a modular design.

Key words: distributed smart cameras, smart camera middleware, agent-oriented middleware, sensor fusion

1 Introduction

Smart cameras have been the subject of study in research and industry for quite some time. While in the "early days" sensing and processing capabilities were very limited, we have seen a dramatic progress in smart camera research and development in the last few years [1,2,3]. Recently, much effort has been put in the development of networks of smart cameras. These *distributed smart cameras (DSC)* [4,5,6] are real-time distributed embedded systems that perform computer vision using multiple cameras. This new approach is emerging thanks to a confluence of demanding applications and the huge computational and communications abilities predicted by Moore's Law.

[*] Corresponding author.
Email addresses: `Bernhard.Rinner@uni-klu.ac.at` (Bernhard Rinner), `Markus.Quaritsch@uni-klu.ac.at` (Markus Quaritsch).

While sensing, processing and communication technology is progressing at high pace, we unfortunately do not experience such a rapid development on the system-level software side. Designing, implementing and deploying applications for distributed smart cameras typically is a complex and challenging endeavor. So we would like to get as much support as possible from system-level software on the DSC network. Such a system-level software or *middleware system* abstracts the network and provides services for the application. As we will see later in this chapter a DSC network has a significantly different characteristic compared to other well-known network types such as computer networks [7] or sensor networks [8]. Thus, we can not directly adopt middleware systems available for these networks.

From the application's point of view, the major services a middleware system should provide are for the distribution of data and control. However, DSC networks are mostly deployed to perform distributed signal processing applications. Thus, middleware systems for DSCs should also provide dedicated services for these applications. Our focus lies on support for distributed image processing and sensor fusion.

In this chapter we introduce our approach towards a middleware system for distributed smart cameras. We first present a brief overview of smart camera architectures and distributed smart camera networks. In Section 4 we focus on embedded middleware systems where we start with an introduction of a generic middleware architecture and middleware systems for general-purpose networks. Middleware systems for embedded platforms as well as the differences for DSC networks are also covered in this section. Section 5 presents our agent-based middleware approach for distributed smart cameras. In Section 6 we describe the implementation of our middleware system and present two cases studies, i.e., a decentralized multi-camera tracking application and sensor fusion case study. Section 7 concludes this chapter with a brief discussion.

2 Smart Cameras

A generic architecture of a smart camera comprised of a sensing, processing and communication unit is depicted in Figure 1. The image sensor, which is implemented either in CMOS or CCD technology, represents the data source of the processing pipeline in a smart camera. The sensing unit reads the raw data from the image sensor and often performs some preprocessing such as white balance and color transformations. This unit also controls important parameters of the sensor, e.g., capture rate, gain, or exposure, via a dedicated interface. The main image processing tasks take place at the processing unit which receives the captured images from the sensing unit, performs real-time image analysis and transfers the abstracted data to the communication unit.

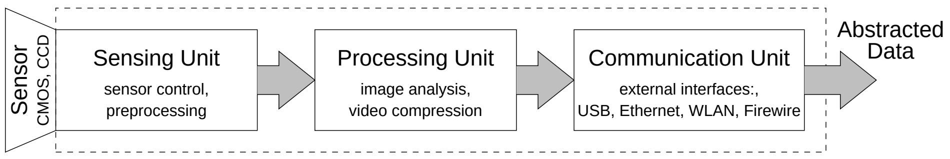

Sensor
CMOS, CCD
Sensing Unit
sensor control, preprocessing
Processing Unit
image analysis, video compression
Communication Unit
external interfaces:, USB, Ethernet, WLAN, Firewire
Abstracted Data

Fig. 1. A generic architecture of a smart camera

The communication unit controls the whole processing pipeline and provides various external interfaces such as USB, Ethernet or Firewire.

These generic units are implemented on various architectures ranging from system-on-chip (SoC) platforms over single processor platforms to heterogeneous multi-processor systems. Field programmable gate arrays (FPGAs), digital signal processors and/or microprocessors are popular computing platforms for smart camera implementations. The main design issues for building smart cameras are to provide sufficient processing power and fast memory for processing the images in real-time while keeping the power consumption low.

Smart cameras deliver some abstracted data of the observed scene. It is natural that the delivered abstraction depends on the camera's architecture and application and almost every smart camera currently delivers a different output. Smart cameras perform a variety of image processing algorithms such as motion detection, segmentation, tracking, object recognition and so on. They typically deliver color and geometric features, segmented objects or rather high-level decisions such as wrong way drivers or suspect objects. The abstracted results may either be transferred within the video stream, e.g., by color-coding, or as a separate data stream. Note that the onboard computing infrastructure of smart cameras is often exploited to perform high-level video compression and only transfer the compressed video stream.

3 Distributed Smart Cameras

Visual sensors provide a huge amount of data on a single scene. However, in many cases a single view is not sufficient to cover a certain region of interest. Parts of a scene may be occluded due to some spatial constraints and the area covered by a single camera is also very limited. Therefore, multiple camera installations are required to observe a certain region of interest. By having different views, distributed vision has the potential to realize many more complex and challenging applications than single camera systems.

Most installations today follow a centralized architecture where huge amounts of processing power is provided in the back office for image processing and scene analysis (e.g. [9,10]). But processing the images of multiple sensors on a central host has several drawbacks. First of all, the communication costs are very high. Using analog CCTV cameras requires dedicated high-bandwidth wiring from each camera to the back office and digitalization of the analog images. But also digital cameras connected via standard Ethernet and communicating via the Internet protocol (IP) require plenty of bandwidth to transfer the raw images. Encoding the video data is often not an option because of the loss in quality and added artifacts which renders the decoded images useless for further analysis. Another issue of centralized systems is scalability. The main limiting factors are (1) communication bandwidth that can be handled in the back office, and (2) processing power required for analyzing the images of dozens of cameras.

Smart cameras are key components for future distributed vision systems and promise to overcome the limitations of centralized systems. Distributed computing offers greater flexibility and scalability than centralized systems. Instead of processing the accumulated data on a dedicated host, scene analysis is done in a distributed manner within the smart camera network. Individual cameras, therefore, have to collaborate on certain high-level tasks (e.g. scene understanding, behavior analysis). Low-level image processing is done on each camera. Collaboration among cameras is founded on abstract descriptions of the scenes provided by individual cameras.

Fault tolerance is another aspect in favor of a distributed architecture. Since the reliability of a centralized system depends on a single or a few components in the back office, the whole system may break due to a failure in a single component. Distributed smart cameras, in contrast, may degrade gracefully. If a single camera fails, a certain view of the scene is not available; but the other cameras may compensate failures of single components.

The distributed architecture also influences the communication infrastructure. Centralized systems demand high-bandwidth links from the cameras to the central processing host. Smart camera networks, in contrast, communicate in a peer-to-peer manner and the bandwidth requirements are also significantly lower because instead of raw image data only abstract information is exchanged. This further allows to incorporate cameras that are connected wirelessly. However, in some application domains, e.g. video surveillance, it is still required to archive the acquired video footage which is typically done on a central storage server. But this demands significantly less bandwidth compared to systems based on centralized processing since the archived video is usually of lower resolution and lower frame-rate. Moreover, archiving is done only in case of certain events for a short period of time.

The development of distributed smart cameras poses several new challenges. Each camera observes and processes the images of a very limited area. High-level computer vision algorithms, however, require information from a larger context. Hence, the important part is to partition the algorithms so that intermediate results can be exchanged with other cameras in order to work collaboratively on certain tasks. Lin et al. [11] describe the partitioning of their algorithm for gesture recognition to be used in a peer-to-peer camera network. Either the segmented image, contour points, or ellipse parameters after an ellipse fitting step can be used to recognize the gesture when a person is observed by two cameras. Collaboration can also be done at lower levels. Dantu and Joglekar [12] investigated in collaborative low-level image processing, like smoothing, edge detection and histogram generation. Each sensor first processes its local image region and the results are then merged hierarchically.

Most computer vision algorithms need to know the parameters of the camera. In a smart camera network the cameras also need to know the position and orientation of the other cameras—at least of cameras in the immediate vicinity or cameras observing the same scene—in order to do collaborative image analysis. Calibrating each camera manually is possible (c.f. [13,14]), but this requires a lot of time and effort. Adding a new camera or changing the position or orientation of a camera necessitates to update the calibration. In an autonomous distributed system it would be much more appropriate if the smart cameras are able to obtain their position—at least relative to neighboring cameras—and orientation by observing their environment and the objects moving within the scene (e.g. [15,16,17,18]).

In a smart camera network which consists of dozens to hundreds of cameras it is tedious, if not impossible, to assign each camera certain tasks manually. Often it is necessary to adapt a given allocation of tasks due to changes in the network (e.g. adding or removing cameras) or changes in the environment. It would be more appropriate to assign tasks to the smart camera network as a whole, possibly with some constraints, and the cameras then organize themselves, i.e., form groups for collaboration and assign tasks to certain cameras or groups of cameras. Reconfiguration due to changes in the environment can also be done in a self-organizing manner.

Smart camera networks are basically heterogeneous distributed systems. Each camera comprises different types of processors, and also within the whole network various smart cameras may be deployed. This makes the development of applications for distributed smart cameras very challenging. A substantial system-level software, therefore, would strongly support the implementation [19]. On the one hand, the system-level software has to provide a high-level programming interface for applications executed on the smart cameras. The most important part of the application programming interface is a suitable abstraction of the image processing unit. An application has to be able to interact with the image processing algorithm in a uniform way, regardless of the underlying smart camera platform. On the other hand, the system-level software should simplify the development of distributed applications for smart camera networks. Implementing the networking functionality as part of the system level software is the foundation for collaboration of various applications on different smart cameras.

3.2 Application Development for Distributed Smart Cameras

Developing applications for a network of distributed smart cameras requires profound knowledge in several disciplines. First, algorithms for analyzing image data have to be developed or adapted to specific needs. This requires good knowledge in the domain of computer vision as well as algorithmic understanding.

The next step is to bring these algorithms onto the embedded smart camera platform, often with real-time constraints in mind (e.g., operating at 25 fps), which demands deep knowledge about the underlying hardware and its capabilities as well as the available resources in order to optimize the implementation. A set of algorithms is then selected and encapsulated within the application logic to put together a specific application.

From this description it is obvious, that there are at least three different roles involved during application development. They are: (1) algorithm developer, (2) framework developer (platform expert), and (3) application developer (system integrator).

A solid middleware with well defined interfaces between the application developer and algorithm developer would, therefore, greatly enhance the process of application development for smart camera networks. Reducing the time-to-market and improved software quality are also beneficial consequences.

4 Embedded Middleware for Smart Camera Networks

A middleware is a system-level software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to make it

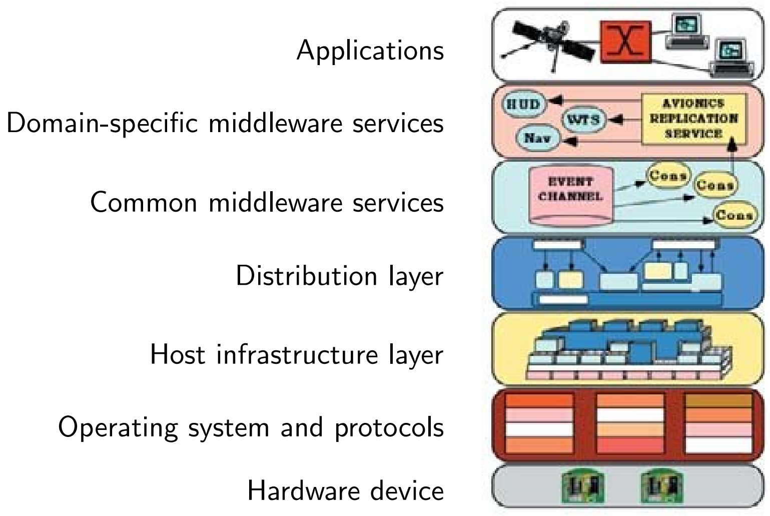

Applications
Domain-specific middleware services
HUD
WTS
Nav
AVIONICS REPLICATION SERVICE
Common middleware services
EVENT CHANNEL
Cons
Cons
Cons
Distribution layer
Host infrastructure layer
Operating system and protocols
Hardware device

Fig. 2. Layers of a general purpose middleware [21].

easier and more cost effective to develop distributed systems [20]. Originally, middleware implementations were targeted for general purpose applications with the primary goal to simplify distributed application development; real-time considerations or resource limitations were not an issue. But nowadays embedded systems make also use of middleware implementations which poses additional constraints and requirements on the design and implementation of the middleware.

4.1 Middleware Architecture

Middleware implementations are usually very comprehensive. They have to run on different hardware platforms, support various communication channels and protocols and also bridge applications running on different platforms, possibly in different programming languages, into a common distributed system. In order to support flexibility of software on different levels, a layered architecture is often used. This is also the case for middleware implementations. A very general partitioning into different layers of abstraction is given by Schmidt [21] (c.f. Fig. 2).

The *operating system* along with its hardware drivers, concurrency mechanisms and communication channels builds the basis of each middleware. It contains drivers for the underlying hardware platform and provides basic mechanisms for accessing the devices as well as concurrency, process and thread management, and inter-process communication.

The *host infrastructure layer* encapsulates the low-level system calls in reusable modules and enhances communication and concurrency mechanisms. Moreover, this layer hides non-portable aspects of the operating system and is the first step towards a portable and platform independent middleware. The interface provided to higher layers is usually object-oriented. Examples for this layer are the Java virtual machine or .NET's common language run-time.

The *distribution layer* integrates multiple hosts in a network to a distributed system and defines higher-level models for distributed programming. The distribution layer enables developers to program distributed applications much like standalone applications. This layer is comparable to Sun's Remote Method Invocation (RMI) in Java or CORBA.

The *common middleware services* layer augments the subjacent distribution layer by defining domain independent components and services which can be reused in applications and thus simplify development. Such components provide, for example, database connection pooling, threading and fault tolerance but also services common for distributed applications, e.g. logging or managing global resources.

The *domain specific* layer provides services for applications of a particular domain, e.g. e-commerce or automation. Services provided by this layer are also intended to simplify application development.

The highest layer of this architecture finally is the *application layer*. Individual applications for a distributed system are implemented using services provided by the lower layers, especially the domain specific layer and the common middleware services layer

4.2 General Purpose Middleware

In general purpose computing, different middleware implementations have evolved during the last decades. Probably the most prominent middleware standard is OMG's Common Object Request Broker Architecture (CORBA) [22]. CORBA is a distributed object system which allows objects on different hosts to interoperate across the network. CORBA is designed to be platform independent and not constrained to a certain programming language. An object's interface is described in a more general language, the interface description language (IDL), which is then mapped to the native data-types of a programming language. While the CORBA specification is very comprehensive and heavy-weighted, Real-Time CORBA (RT-CORBA) and Minimum CORBA have been specified [23,24] for resource constrained real-time systems. Schmidt et al. implemented with "TAO" [25] the RT-CORBA specification.

Another middleware for networked system is Microsoft's Distributed Component Object Model (DCOM) [26,27]. DCOM allows software components to communicate over a network via remote instantiation and method invocations. Unlike CORBA, which is designed for platform and operating system independence, DCOM is implemented primarily on the Windows platform.

Java Remote Method Invocation (RMI) [28] promoted by Sun follows a similar approach. RMI allows to invoke a method of an object in a different Java Virtual Machine, possibly on a different host, and thus simplifies the development of distributed Java applications. Java's integrated object serialization and marshaling mechanism allows to use even complex objects for remote method invocations. Although RMI is limited to the Java programming language, it is more flexible than DCOM because Java is available for several platforms.

4.3 Middleware for Embedded Systems

Embedded systems are becoming more and more distributed. Hence, some form of middleware would greatly support application development for networked embedded devices. Wireless sensor networks are an inherently distributed system where individual sensors have to collaborate. But the resources and capabilities of the individual sensors are very limited; typically only a couple of scalar values are sensed.

The requirements on a middleware for wireless sensor networks are also significantly different compared to those in general purpose computing. These middleware systems focus on reliable services for ad-hoc networks and energy awareness [29].

Molla and Ahmed [30] survey recent research on middleware for wireless sensor networks. Most implementations are based on TinyOS [31], a component-oriented, event-driven operating system for sensor nodes (motes). Several interesting approaches have been implemented and evaluated. The spectrum ranges from a virtual machine on top of TinyOS, hiding platform and operating system details, to more data-centric middleware approaches for data aggregation (shared tuplespace) and data query. Agilla [32] and In-Motes [33], for example,use an agent-oriented approach. Agents are used to implement the application logic in a modular and extensible way and agents can migrate from one mote to another. Cougar [34] or TinyDB [35] follow the data-centric approach, integrating all nodes of the sensor network into a virtual database system where the data is stored distributed among several nodes.

4.4 *Differences for Distributed Smart Cameras*

Compared to the middleware systems described up to now, a middleware for distributed smart cameras has to fulfill significantly different requirements. This is merely due to different resource constraints but also a consequence of the application domain of smart camera networks.

In general purpose computing, platform independence is a major issue. Hence, several layers of indirection encapsulate these platform dependencies and provide high-level interfaces. General purpose middleware implementations are, therefore, rather resource consuming and introduce a noticeable overhead. Wireless sensor networks, on the other hand, have very tight resource limitations in terms of processing power and available memory and middleware implementations have to cope with this circumstances. Typical embedded smart camera platforms, as presented in Section 2, lie in-between general purpose computers and wireless sensor nodes when considering the available resources. Hence, a middleware for smart camera networks has to find a trade-off between platform independence, programming language independence and the the overhead introduced by the middleware.

Distributed smart cameras are intended for processing the captured images close to the sensor which requires sophisticated image processing algorithms. Support from the middleware is necessary in order to simplify application development and integration of image processing tasks into the application . Moreover, monitoring the resources used by the individual image processing algorithms is required.

Communication in wireless sensor networks is relatively expensive compared to processing and thus only used sparingly, e.g., in case of certain events or to send aggregated sensor data to a base station. Collaboration of individual nodes is typically inherent to the application. Smart camera networks demand higher communication bandwidth for sending regions of interest, exchanging abstract features extracted from the images, or even streaming the video data.

Typical surveillance applications of camera systems comprise several hundreds up to thousands of cameras (e.g. on airports or train stations) and a plethora of different tasks have to be executed. Assigning those tasks manually to the cameras is almost impossible. Hence, the idea is that a user simply defines a set of tasks that have to be fulfilled (e.g. motion detection, tracking of certain persons) together with some restrictions and rules what to do in case of an event. The camera system then allocates the tasks to cameras itself, taking into account the restrictions imposed by the user. In some cases a task can not be fulfilled by a single camera. Individual cameras, therefore, have to organize themselves and collaborate on a certain task. Having a single point of control in such a self-organizing system is discouraged, striving for a distributed and decentralized control.

According to the discussion of the different requirements it is obvious that a different kind of middleware is necessary. Middleware for wireless sensor networks is not intended to cope with advanced image processing tasks and sending large amounts of data. Adapting general purpose middleware, on the other hand, would be feasible and able to fulfill the given requirements, but the introduced overhead would not yield in an efficient resource utilization.

5 The Agent-oriented Approach

Agent oriented programming (AOP) has become more and more prominent in software development during the last years. The agent oriented programming paradigm extends the well known object oriented programming paradigm and introduces active entities, so called *agents*.

This section gives a short introduction of mobile agent systems. Furthermore, the use of mobile agents in embedded devices and especially distributed smart cameras is discussed.

5.1 From Objects to Agents

Agent systems are a common technology for developing general purpose distributed applications. The agent-oriented paradigm is used in many application domains such as electronic commerce [36,37], information management [38] or network management [39,40] to name just a few. Although agents are used in various domains, there exists no common definition of the term *agent*. Possibly, the widespread use of agent-oriented programming makes a definition difficult. Agents are ascribed different properties, depending on their use. However, for the further considerations, a rather general definition, adopted from [41], is used:

> *An agent is a software entity that is situated in some environment and that is capable of autonomous actions in this environment in order to meet its design objectives.*

The most important property of an agent, and this is common for all uses of agents, is autonomy. This is also the fundamental distinction from the object-oriented programming (OOP) paradigm. Agent-oriented programming can be seen as an extension of object-oriented programming. In object-oriented programming, the main entity is an object. Objects are used to represent logical entities or a real-world object. An object consists of an internal state, stored in member-variables, and corresponding methods to manipulate the internal state. Hence, objects are passive entities as their actions have to be triggered from outside. Agents extend objects as they are capable to perform autonomous actions. In other words, agents can be described as pro-active objects.

As stated in the definition above, agents are situated in an environment. This environment is usually called *agency*. The agency provides the required infrastructure for the agents. This includes communication and a naming service, for example. Some requirements for an agency are discussed in [42]. Using a well-defined agency guarantees that agents are able to interact with their outside world as well as other agents in a uniform manner. A minimal set of services is, therefore, defined in the MASIF standard [43,44] and the FIPA ACL [45] standard. When an agency conforms to one of these standards it ensures interoperability with other agencies also complying to the same standard.

Practical applications of agent systems consist of a couple of agents, possibly up to several dozens, distributed among several environments in a network. The agents can communicate with each other, regardless of the environment they inhabit and they may collaborate on a certain task. Agent systems are thus perfectly qualified for distributed computing. The data as well as the computation is distributed among the network by using agents. Moreover, if it is not desired nor feasible to have a central point of control in a system, agents can also be used to realize decentralized systems where control is spread among several hosts in a network.

5.2 Mobile Agents

Agents, as described up to now, reside on the same agency during their lifetime. But enhancing the agent-oriented approach with mobility makes this paradigm much more powerful and also more flexible since they are able to move from one agency to another. Making agents mobile allows in certain situations to significantly reduce the network communication. Furthermore, the execution time of certain tasks can be reduced by exploiting the mobility of agents. If, for example, an agent requires a specific resource that is not available on its current host, the agent has two options: either the agent uses remote communication to access the resource or the agent migrates to the host on which the resource is available. Which option to choose depends on whether accessing a resource remotely is possible and also on the costs of remote interaction versus local interaction.

Prominent representatives of mobile agent systems are D'Agents [46], Grasshopper [47], Voyager [48], and Diet-Agents [49,50], among others.

5.3 Code Mobility and Programming Languages

Concerning the implementation of mobile agent systems, the agent's mobility property incurs some requirements on the programming language. Agent systems are typically used on a variety of hardware platforms and operating systems; even within a single network different kinds of hosts may be present. Mobile agents, however, must be able to migrate to any host within the system which means that the agent's code has to be executable on all these hosts. Therefore, early mobile agent systems were implemented using scripting languages while in the last couple of years languages based on intermediate code, i.e. Java and .NET, are preferred. Programming languages that are compiled to native code such as C or C++ are hardly used for implementing mobile agent systems.

Scripting languages allow to execute the same code on different platforms without the need of recompilation. Platform independency is realized by providing an interpreter or an execution environment which executes the code. As a consequence, the performance of interpreted code is not convincing. Java and .NET overcome these limitations by exploiting just-in-time compilers that generate native code before execution which brings a significant performance shift.

Another aspect of mobile agents is the migration of agents from one agency to another. This basically requires the following steps:

- Suspend the agent and save its current internal state and data
- Serialize the agent (data, internal state and possibly code)
- Transfer the serialized agent and its data to the new host
- Create the agent from its serialized form
- Resume agent execution on the new host

Suspending an agent and resuming it on another host is not trivial without the cooperation of the agent. Hence, two types of migration are distinguished: (1) weak migration, and (2) strong migration [51].

Strong migration, also called transparent migration, denotes the ability to migrate both the code and current execution state to a different host. Strong migration is supported by just a few programming languages and thus agent platforms (e.g. Telescript [52]); Java and .NET are not among these.

Weak migration, or non-transparent migration, in contrast, requires the cooperation of the agent. The agent has to make sure to save its execution state before migrating to another host and resume the execution from its previously saved state after the migration. Weak migration is supported by all mobile agent systems.

5.4 Mobile agents for Embedded Smart Cameras

Although agent-oriented programming is widely used in general-purpose computing for modeling autonomy and goal-oriented behavior, it is rather uncommon for embedded systems. The reasons therefore are the significantly different requirements of software for embedded devices. Resources such as memory and computing power are very limited; and often embedded systems have to fulfill real-time requirements. The basically non-deterministic behavior of autonomous agents further hinders their use in embedded systems.

Nevertheless, research has shown that the agent-oriented programming paradigm can also enhance software development for embedded systems. Applications include, among others, process control, real-time control, and robotic (c.f. [53,54,55]). Stationary agents are used to represent individual tasks within the system. The communication structure of agents (i.e. which agents communicate with one another) is typically fixed according to the physical circumstances they are used to model and does not change over time.

Smart cameras are also embedded systems and mobile agents are perfectly suited to manage whole networks of smart cameras. The ultimate goal is that smart camera networks operate completely autonomously with no or only minimal human interaction. For example, a smart camera network controls access to a building and identifies all persons entering the building. When an unknown person enters the building, the position of the person is tracked and security staff gets informed. The agent-oriented paradigm can be used to model individual tasks within the whole system, e.g. face recognition or tracking. By agent communication tasks on different cameras can collaborate in order to work jointly on a certain mission.

However, one aspect against using mobile agent systems on embedded smart cameras is the overhead introduced by commonly used programming languages, i.e. Java and .NET (c.f. Section 5.3). But this does not necessarily mean that it is not possible to implement mobile agent systems more efficiently and more resource-aware and thus also applicable on embedded systems. Mobile-C [56], for example, is a mobile agent system implemented in C/C++. The development of Mobile-C is motivated by applications that require direct low-level hardware access. Moreover, this agent system conforms to the FIPA agent standard and further extends this standard to support the

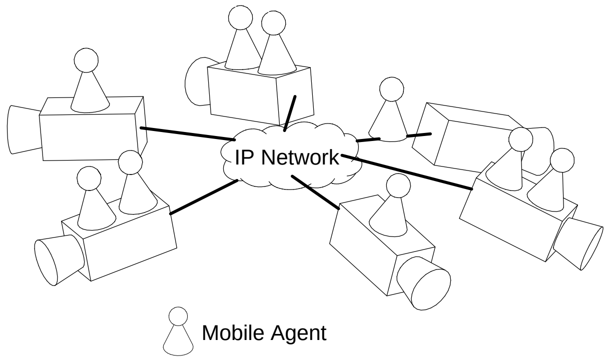


Fig. 3. Architecture of a smart camera network with a mobile agent system.

mobility of agents. Although the chosen programming language is C/C++, the agent code is interpreted using the *Ch* C/C++ interpreter [57]. The task of an agent is divided into multiple sub-tasks which are organized in a task list. Upon migration, the next task in the list will be executed.

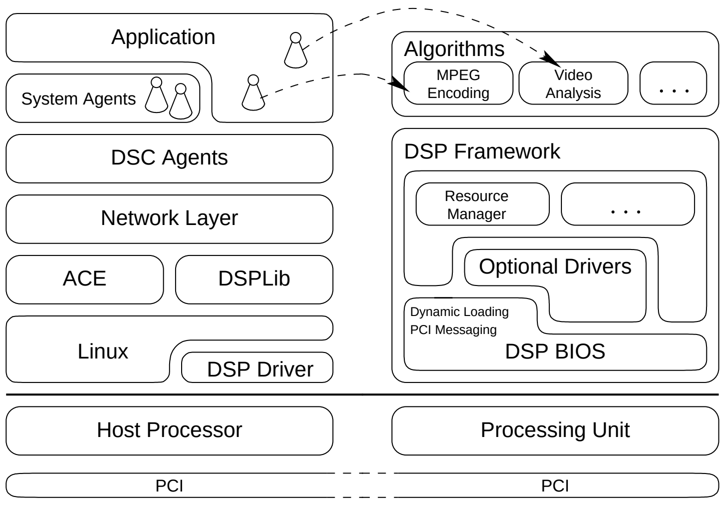
6 An Agent-System for Distributed Smart Cameras and its Application

The feasibility and applicability of the agent-oriented approach on embedded smart cameras, is presented on two case studies, namely autonomous and decentralized multi-camera tracking, and sensor fusion. In both case-studies we focus on the middleware services that simplify application development. But first of all, a description of our agent system, *DSCAgents*, is given.

6.1 DSCAgents

DSCAgents is an agent system designed for smart camera networks and basically suited for different hardware architectures. The design is founded on the general architecture as described in Section 2, which consists of a communication unit and a processing unit as well as one or more image sensors. The operating system on the smart cameras is assumed to be an embedded Linux but other POSIX-compliant operating systems are also feasible. For efficiency reasons, the chosen programming language is C++, which also influences the design to some degree. The concrete implementation targets the *Smart**Cam*** platform developed by Bramberger et al. [3].

The overall architecture of a smart camera network and the mobile agent system is depicted in Figure 3. The smart cameras are connected via wired (or possibly wireless) Ethernet whereas each camera hosts an agency, the actual



(a) Host processor (b) Processing unit

Fig. 4. Software Architecture.

run-time environment for the mobile agents. The agency is situated on the general purpose processor of the communication unit. Mobile agents represent the image processing tasks that have to be executed on the smart camera network.

6.1.1 Software Architecture

Since the underlying hardware architecture is in general a multi-processor platform, the software architecture has to reflect this. The host processor handles the communication tasks and thus executes the agent system while the processing unit is devoted to image processing. Figure 4 depicts the software architecture of our middleware.

The software architecture on the communication unit can be partitioned into layers as described in Section 4.1. On the processing unit the Linux kernel is used as operating system. It already comprises device drivers for a great number of hardware components, most important networking and busses connecting further components. A custom device driver is used to manage the digital signal processors (DSPs) on the processing unit and exchange messages between the processors.

The host abstraction layer basically consists of the Adaptive Component Environment (ACE), and the *Smart**Cam*** framework. ACE [58] is a reusable C++ framework that provides a portable and light-weight encapsulation of several communication mechanisms, network communication as well as inter- and intra-process communication. The DSPLib provides an object-oriented interface to the processing unit which allows to load and unload executables on the processing unit as well as exchanging messages between algorithms on the processing unit and applications on the host processor.

The network layer is based on the ACE framework and basically provides mechanisms to establish network connections between hosts and asynchronous message-oriented communication. Different low-level protocols are supported. *DSCAgents* finally matches the common middleware layer. It provides a runtime environment for the agents, manages the agent's life-cycle, and allows agents on different agencies to communicate with each other.

A number of stationary system agents provide additional services required for building an application. The *NodeManagementAgent* is concerned with all management tasks on a smart camera. This agent is available under a well-known name and can also be accessed remotely. Services provided by this agent are agent creation (local as well as remote), monitoring available resources, and get information about agents in an agency, among others. The *SceneInformationAgent* manages all information regarding the vision system. Depending on the actual deployment this may include camera calibration properties, position and orientation of the camera, list of neighboring cameras, and visual properties (e.g. image resolution, color-depth). The *ImageProcessingAgent* is the central instance for interacting with the camera's processing unit. This agent is only locally accessible, preventing remote execution of image processing tasks. The main functionality includes loading an unloading image processing algorithms, messaging between agents and algorithms, as well as providing information on the available resources.

On the processing unit, the *DSPFramework* [59,3] is the foundation for the algorithms running on the DSPs. The operating system used is DSPBios, provided by Texas Instruments, which is enhanced with dynamic loading capabilities and inter-processor messaging. Additional modules, such as optional drivers or the resource manager, can be loaded dynamically during runtime. Also the algorithms executed on the DSPs can be loaded and unloaded dynamically during operation without interrupting the other algorithms.

6.1.2 Mobility of Agents

DSCAgents supports mobility of agents, i.e., agents can move from one camera to another. Unfortunately, transparent migration is hard to implement in C++, so we chose weak migration based on remote cloning. Agent migration, thus, involves the following steps:

(1) The originating agent saves its internal state in a serializable form.
(2) A new agent is created on the destination agency.
(3) The new agent initializes itself with the initial state from the originating agent.
(4) The originating agent is destroyed.

The steps (1) and (3) require the cooperation of the agent, i.e., have to be implemented by the application developer. The other steps, (2) and (4), are handled by the agent system. It is not mandatory that the agent's code is available on the destination agency. In case it is not available the agent code can be loaded as a dynamic library which is provided in an agent repository in the network. This allows to deploy new types of agent during operation of the camera network.

An agent comprises the image processing algorithms in form of a dynamic executable which is loaded and unloaded onto the image processing unit as needed. Hence, the image processing algorithms are also flexible and can be modified during runtime; even new image processing algorithms can be added to the system after deployment.

6.1.3 Evaluation

DSCAgents' implementation targets embedded systems and thus has to use the scarce resources such as memory and processing power sparingly. Table 1 lists the code-size of *DSCAgents* and its modules (stripped, cross-compiled binaries). The total consumption of permanent memory, including all libraries, is less than 3.5 MB. Compared to other agent systems, such as DietAgents, Grasshopper or Voyager, which require 20 MB and more (this includes the Java virtual machine and the classpath), *DSCAgents* is fairly lightweight. A large portion of the codesize is contributed by the ACE library which not only abstracts networking functionality but contains several other components. When only the networking capabilities are required, an optimized version of this library may be compiled without the unnecessary components which further decrease the codesize.

DSCAgents requires approximately 2.5 MB of RAM after startup. When creating additional agents, the memory consumption increases: an agency comprises 50 agents—which is a fairly large number when thinking of smart camera networks—the memory consumption goes up to 3.3 MB. Of course, this heavily depends on how much memory the agent itself allocates. But for the purpose of this evaluation an agent was used that allocates no additional memory. Note that in this case each agent has its own thread of execution which accounts for the greater part of the allocated memory.

Table 2 summarizes the execution times for agent creation, agent migration and loading an image processing algorithm. Creating a new agent on our camera takes about 4.5 ms and 10.1 ms creating the agent locally and on a remote agency, respectively. Regarding mobility of agents it is interesting to know, how long it takes to move an agent from one camera to another. Since *DSCAgents* uses remote cloning, the time for agent migration is the same as

Libraries		2680 MB
ACE	2060 kB	
Boost	442 kB	
SmartCam Framework	178 kB	
DSCAgents		885 kB
Total		3565 kB

Table 1
Memory consumption.

		Time
Creating an agent	(local)	4.49 ms
	(remote)	10.11 ms
Loading image processing algorithm		17.86 ms

Table 2
Execution times for frequent operations measured on the embedded smart camera.

for remote agent creation. Loading an image processing algorithm from an agent takes approximately 18 ms. This includes also the time required to send the executable through the DSPAgent to one of the DSPs.

Figure 5 investigates in agent communication and shows the average transmission time against the message size. Note the logarithmic scale of the x- and y-axis. The presented values are the average of 20 runs. Messaging between agents on the same camera can be done very fast and is independent of the message size because this does not require to send the message over the network. Messaging between agents on different hosts is somewhat higher and also depends on the message size. For small messages (about 1000 Byte, the interface's MTU) the messaging times are nearly constant. As the message size increases, it also takes longer until the sending agent receives an acknowledgment. The time required for larger messages goes linear with the message size.

6.2 Decentralized Multi-camera Tracking

We have implemented an autonomous decentralized tracking method that follows the so-called tag-and-track approach. This means that not all moving objects within the monitored area are tracked but only a certain object of interest. Furthermore, the tracking task is only executed on the camera that currently sees the target while all other cameras are not affected. The basic idea is to virtually attach a tracking instance to the object of interest. The

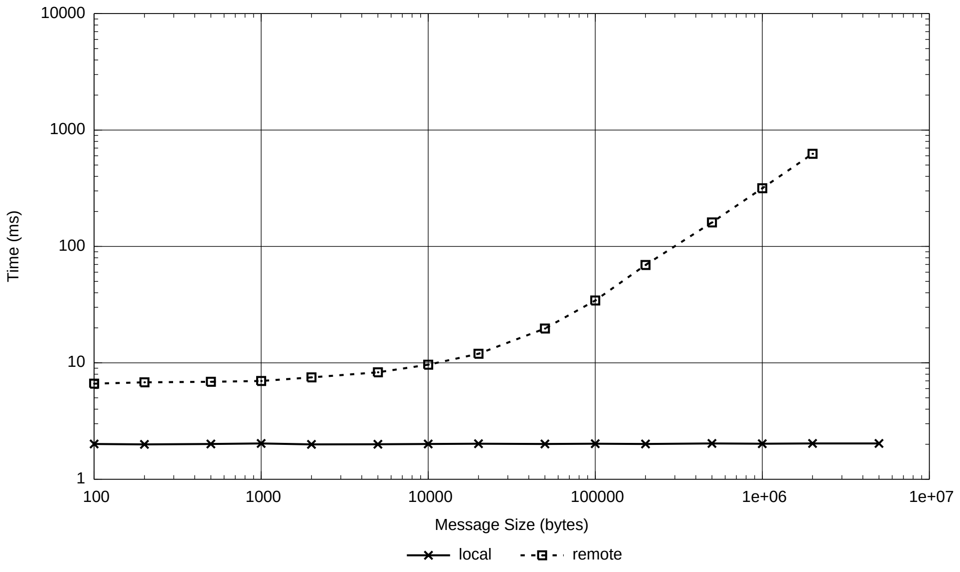


Fig. 5. Average messaging time.

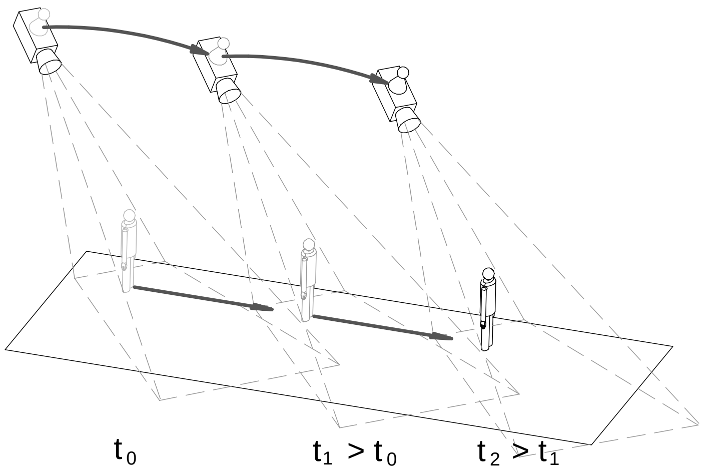


Fig. 6. Basic idea of decentralized object tracking.

tracking instance (i.e. the agent) then follows its target in the camera network from one camera to another. Mobility of agents inherently supports this highly dynamic tracking approach and furthermore allows to use different tracking and handoff strategies as well as different tracking algorithms, even after deployment of the whole system. Figure 6 illustrates the position of the target together with its tracking instance over time. A more elaborate discussion is given in [60]

6.2.1 Tracking Application

Following the target from one camera to the next is done without a central control instance. Moreover, the camera topology is stored in a distributed manner on the camera's SceneInformationAgent. Neighborhood relations are represented by so-called *migration regions*. A migration region is basically a polygon in 2D image coordinates that has assigned a list of neighboring cameras and a motion vector for distinguishing different directions. Hence, the migration regions define a directed graph representing the neighborhood relations among cameras.

The tracking instance is made up of a tracking algorithm for following the position of a moving object in a single view and a mobile agent containing the application logic. A strict separation between tracking algorithm and application logic allows on the one hand to equip an agent with different algorithms for object tracking, depending on the application or environmental conditions. On the other hand, different strategies for following an object within the smart camera network can be implemented using the same tracking algorithm.

6.2.2 Target Handoff

The most crucial part in our multi-camera tracking approach is the target handoff from one camera to the next. The handoff requires the following basic steps:

(1) Select the "next" cameras where the target may appear next
(2) Migrate the tracking instance to the next cameras
(3) Re-initialize the tracking task
(4) Re-detect the object of interest
(5) Continue tracking

Identifying the potential next cameras for the handoff uses the neighborhood relations as discussed previously. The next two steps of the handoff procedure are managed by the mobile agent system. Object re-detection and tracking are then continued on the new camera.

The tracking agent may use different strategies for the target handoff [61]. We use the master/slave approach which has the major benefit that the target is observed as long as possible. During handoff, there exist two or more tracking instances dedicated to one object of interest. The tracking instance that currently has the target in its field of view is called *master*. When the target enters a migration region, the master initiates the creation of the slaves on all neighboring cameras. The slaves in turn re-initalize the tracking algorithm with the information passed from the master and wait for the target to appear. When the target enters the slave's field of view, it becomes the new master while the old master and all other slaves terminate. A sophisticated handoff protocol is used to create the slaves on the neighboring cameras, elect the new master and terminate all other tracking instances after successful target handoff. The handoff protocol minimizes the time to create the slaves and also handles undesired situations (e.g. the target does not appear on a neighboring camera, or more than one slaves claim to have the target detected).

The presented tracking approach has been implemented and tested on tracking a person in our laboratory. Figure 7 illustrates target handoff from one camera to another, showing the view of both cameras in the upper part of the screenshots and the agents residing on the camera in the lower part. Note that the views of both cameras overlap which is not mandatory. The highlighted square in the camera's view denotes the position of the tracked person and the rectangles on the left and right illustrate the migration regions of camera A and B, respectively. Tracking the person is started on camera A (cf. Figure 7a). During handoff (Figure 7b) two tracking instances are present, one on each camera (the highlighted rectangle in the lower part of the screenshot represents the tracking agent). After the handoff, camera B continues tracking the person (cf. Figure 7c).

6.2.3 Evaluation

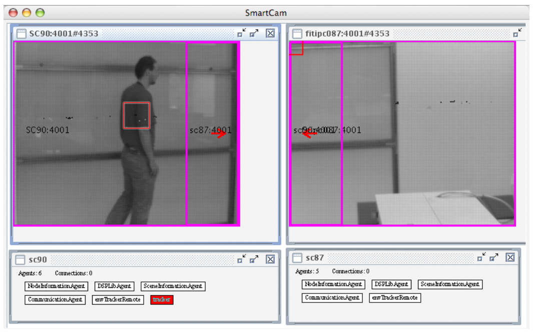
When following an object in a smart camera network, the migration times of the tracking agent are very critical. If the handoff is too slow, it may occur that the tracker is not able to find the target any more or that the target has already left the destination camera's field of view. Hence, the duration for target handoff is subject of evaluation.

During target handoff, three time intervals can be identified. These time intervals have been quantified and summarized in Table 3a. When the tracked object enters the migration region, it takes about 18 ms to create the slave agent on the next camera. Starting the tracking algorithm on the DSP requires 24 ms. This includes loading the dynamic executable to the DSP, starting the tracking algorithm and reporting the agent that the tracking algorithm is ready to run. Initializing the tracking algorithm by the slave agent using the information obtained from the master agent takes 40 ms. Hence, the total migration time is about 80 ms.

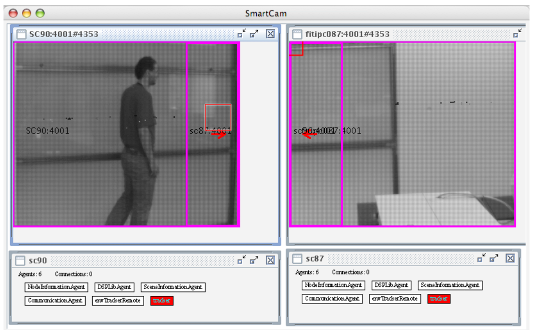
In case a migration region points to more than one neighboring cameras, a slave has to be created on each neighboring camera. The master creates all its slaves almost in parallel exploiting asynchronous communication. Nevertheless, the number of slaves slightly increases the handoff times (cf. Table 3b).

6.2.4 Middleware Support

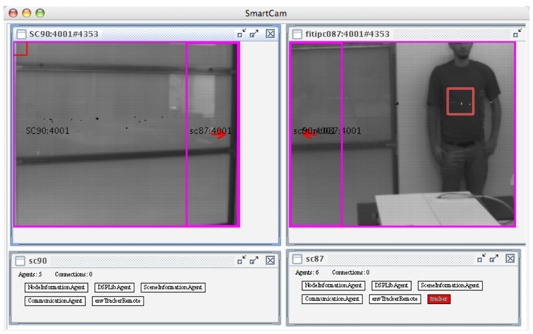
For the multi-camera tracking application the following middleware services are helpful and necessary:



(a) Tracker on camera A



(b) Handover to camera B: the person is in the migration region (highlighted square)



(c) Tracker on camera B

Fig. 7. Visualizing tracking a person among two cameras. Note that he acquired image of a camera and the current position of the person are updated at different rates. Hence, in image (b) the highlighted position is correct while the background image is inaccurate.

Mobility. Tracking agents are, due to their mission, inherently highly mobile. They follow their target from one camera to the next. Hence, the middleware has to support this mobility and providing agents a fast and reliable migration mechanism. *DSCAgents* supports mobility of agents as general service by means of remote cloning (cf. Section 6.1.2).

Dynamic task loading. A consequence of the chosen tag-and-track approach is that only the camera which currently observes the object has to execute the tracking algorithm and the agent decides which tracking algorithm to use. Hence, the tracking algorithm is loaded by the agent on the camera

Creating slave on neighboring camera:	18 ms
Loading dynamic executable:	24 ms
Reinitializing tracking algorithm on slave camera:	40 ms
Total	82 ms

(a) Handoff to a single camera

Number of neighbors	*Time*
1	82 ms
2	98 ms
3	105 ms
4	126 ms

(b) Handoff to multiple neighbors

Table 3
Evaluation of the handoff times

when the agent arrives and unloaded on agent departure. Object tracking has tight timing constraints, even during target handoff. Hence, loading image processing tasks has to be considerably fast.

Dynamic task loading is considered as domain-specific service that is provided by the *ImageProcessingAgent*. This agent handles the whole communication with the image processing part and also supports to load dynamic executables together with the framework on the image processing unit (cf. Figure 4).

Neighborhood relations. In order to follow an object over the camera network, it is crucial to know the position and orientation of the cameras within the network. The middleware, therefore, has to manage and update information on the camera topology, preferably autonomously and in a distributed manner to keep the whole system fault tolerant.

Neighborhood relations are also a domain-specific service which is provided by the *SceneInformationAgent*. For our evaluation, the agent reads the configuration from a file. But in a real-world deployment this agent should observe the activities in the scene and learn the camera parameters automatically.

6.3 Sensor Fusion

The next step in distributed smart cameras is to use not only visual sensors but also integrate other kinds of sensors like infrared cameras, audio sensors or induction loops, among others. The intention is to get more reliable and more robust data while reducing ambiguity and uncertainty. But in order to make advantage of these different sensors it is necessary to correlate the data somehow, i.e. fuse the information from individual sensors.

A lot of research has been conducted over the last decades in sensor fusion. Several data fusion algorithms have been developed and applied, individually and in combination, providing users with various levels of informational details. The key scientific problems, which are discussed in the literature [62,63], can also be addressed to the three fusion levels:

Raw-data Fusion: The key problems, which have to be solved at this level of data abstraction, can be referred to as (i) data association and (ii) positional estimation [64]. Data association is a general method of combining multi-sensor data by correlating one sensor observations set with another set of observations. Common techniques for solving the positional estimation problem are focused on Kalman filtering and Bayesian methods.

Feature Fusion: These approaches are typically addressed by (i) Bayesian Theory and (ii) Dempster-Shafer Theory. Bayesian Theory is used to generate a probabilistic model of uncertain system states by consolidating and interpreting overlapping data provided by several sensors [65]. Bayesian theory is limited in its ability to handle uncertainty in sensor data. Dempster-Shafer theory is a generalization of Bayes reasoning that offers a way to combine uncertain information from disparate sensor sources. Recently, methods for statistical learning theory, i.e., support vector machines [66] have been successfully applied to feature-level fusion

Decision Fusion: Fusion at the decision level combines the decisions of independent sensor detection/classification paths by Boolean operators or by a heuristic score (e.g., M-of-N, maximum vote or weighted sum). The two basic methods for making classification decisions are hard decisions (single, optimum choice) and soft decision in which decision uncertainty in each sensor chain is maintained and combined with a composite measure of uncertainty. There are a few investigations undertaken on level three data fusion in the literature.

6.3.1 The Fusion Model

In the I-SENSE project [67] we investigated in distributed sensor fusion performed in a network of embedded sensor nodes. Our fusion model supports fusion at multiple levels, i.e., raw-data fusion, feature-based fusion, and decision fusion. Moreover, the fusion model considers the data flow in the sensor network as well as the resource restrictions on the embedded sensor nodes.

The fusion model describes the functionality of the distributed fusion application and consists basically of a set of communicating tasks located on different nodes within the network. A directed acyclic graph G is used to represent the

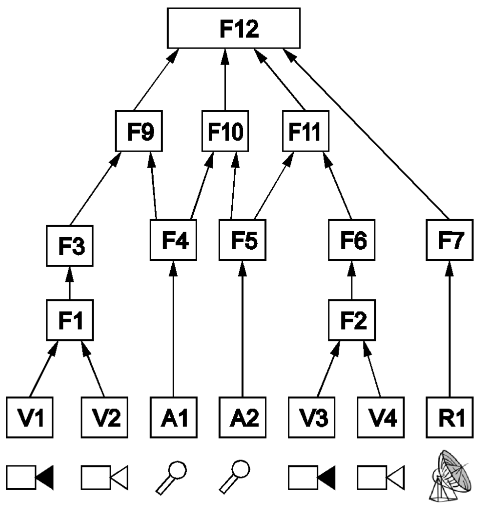


Fig. 8. Example of a simple fusion model.

fusion tasks (nodes) as well as the data flow between those tasks (edges). Each node has some properties describing the resource requirements of the task and the edges also indicate the required communication bandwidth between two tasks.

Figure 8 depicts a simple example of a fusion model. The *sensor tasks* (e.g. V1, . . . ,V4, A1, A2 in Figure 8) always build the bottom layer of the fusion tree. These tasks acquire the data from the environment, independent from other tasks. *Fusion tasks* (e.g. F1 and F2 in Figure 8) and *filter tasks* (e.g. F3, F4, F5 in Figure 8) form the higher layers in the fusion tree. Fusion tasks fully depend on data from other fusion tasks or sensor tasks to produce an output. Upon new input data, the fusion task has to ensure the temporal alignment and calculate the output vector. Filter tasks are similar to fusion tasks except they have only one input channel.

Raw data fusion is done in the lower layers of the fusion graph (e.g. F1 or F2) since the input for these fusion tasks is the raw data of the sensor tasks. Feature fusion takes place in the middle layers of the fusion graph; features from other fusion tasks are fused to an overall feature vector. Decision fusion is done in the upper layers of the fusion graph. Either features from a previous fusion stage are used to generate a decision (e.g. classification) or multiple weak decisions are used to calculate more reliable decisions. The border between feature fusion and decision fusion in the fusion graph is not strict but depends on the current application.

We have evaluated the presented fusion model on vehicle classification, exploiting visual and acoustic data. Therefore, we collected a database consisting of about 4100 vehicles which are either large trucks, small trucks and cars.

The vision-only classifier predicts cars very well (95.19%) but it has problems in distinguishing between small trucks and large trucks. Quite similar results are obtained when using acoustic features only. However, fusing data from both sensors increases the classification performance for all three vehicle classes [67].

6.3.2 Middleware Support

In our fusing approach, the individual fusion tasks are distributed among the nodes and can be reallocated from one node to another. Also the communication structure between individual fusion tasks may change over time. Hence, implementing such a general sensor fusion approach benefits from having a substantial middleware which provides the fundamental services.

Sensor interfaces. Each platform is equipped with a different set of sensors. Although the sensing tasks have to know how to interpret the data from the according sensors, the middleware has to provide a general interface for each class of sensors (i.e. visual sensors, audio sensors). Hence, the sensing tasks do not have to cope with low level sensor interaction but can use the according interface provided by the middleware. This also makes the integration of new sensors easier because it is not required to modify each sensing task but only include the device driver into the middleware. The sensor interfaces are part of the framework on the processing unit and are provided by optional drivers. These drivers are loaded by the middleware from the host processor according to the platform's capabilities. Connecting the sensors and the algorithms follows the publisher/subscriber design pattern [68].

Dynamic task loading. In order to build a flexible framework for sensor fusion, it is possible to reallocate a task executed on one node to another. Hence, it is necessary to load and start task dynamically during runtime on the embedded platform without interrupting other tasks but also stop certain fusion tasks. This domain-specific service is provided by the *ImageProcessingAgent* together with the framework on the image processing unit.

Resource monitoring. The resource monitoring has to keep track of all resources consumed by the image processing tasks, in particular memory, processing power, DMA channels, communication bandwidth, among others. Fusion tasks can only be allocated to a certain node if there are sufficient resources available.

Time synchronization. Distributed sensor data fusion implies a uniform time-base for all nodes. Without a system wide synchronized clock it would be impossible to combine results from different sensors. Therefore, the middleware has to provide a service that keeps all the nodes synchronized and also the individual processors on a single platform. A dedicated agent is responsible to provide this service. For our evaluation the network time protocol[1] (ntp) was sufficient, but one may also implement other synchronization mechanisms.

7 Conclusion

In this chapter we investigated in middleware for distributed smart cameras. Smart cameras combine image sensing, considerable processing power and high-performance communication in a single embedded device. Recent research focuses on the integration of several smart cameras into a network and thus building a distributed system devoted to image processing. Middleware services for distributed systems are used in many application domains, ranging from general purpose computing to tiny embedded systems such as wireless sensor networks. But as discussed in this chapter, the requirements on a middleware for distributed smart cameras are considerably different. Hence, a more specialized kind of middleware is required, taking into account the special needs of distributed smart cameras.

We have proposed to use the agent-oriented approach for managing distributed smart cameras and building applications. Mobile agents are autonomous entities that "live" within the network. The autonomous and goal-oriented behavior of agents allows to create self-organizing distributed smart cameras. In two case-studies we demonstrate the agent-oriented approach on decentralized multi-camera tracking and sensor fusion, respectively. Moreover, we discuss important services a middleware has to provide in the application domains of distributed computer vision and sensor fusion.

From our experience, this middleware eases the development of DSC applications in the several ways: First, it provides a clear separation between algorithm implementation and coordination of these algorithms among different cameras. Second, it strongly supports scalability, i.e., to develop applications for a variable number of cameras. Finally, the available resources can be better utilized by dynamic loading and dynamic reconfiguration services which are usually not available on embedded platforms.

However, there are still a lot of open questions for middleware systems for distributed embedded platforms in general. Topics for further research include (i) support for the application development process (development, operation, maintenance, etc.), (ii) resource awareness as well as (iii) scalability and interoperability—just to name a few.

[1] `http://www.ntp.org/`

References

[1] R. Kleihorst, B. Schueler, A. Danilin, Architecture and Applications of wireless Smart Cameras (Networks), in: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2007), Honolulu, Hawaii, USA, 2007.

[2] W. Wolf, B. Ozer, T. Lv, Smart Cameras as Embedded Systems, Computer 35 (9) (2002) 48–53.

[3] M. Bramberger, A. Doblander, A. Maier, B. Rinner, H. Schwabach, Distributed embedded smart cameras for surveillance applications, IEEE Computer 39 (2) (2006) 68–75.

[4] H. Aghajan, R. Kleihorst (Eds.), Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC 07), Vienna, Austria, 2007.

[5] R. Kleihorst, R. Radke (Eds.), Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC 08), Stanford, CA, USA, 2007.

[6] B. Rinner, W. Wolf, Introduction to Distributed Smart Cameras, Proceedings of the IEEE 96 (10), to appear.

[7] A. S. Tanenbaum, M. van Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, 2006.

[8] I. F. Akyildiz, T. Melodia, K. R. Chowdhury, A survey on wireless multimedia sensor networks, Computer Networks 51 (2007) 921–960.

[9] R. Collins, A. Lipton, T. Kanade, A System for Video Surveillance and Monitoring, in: American Nuclear Society 8th Internal Topical Meeting on Robotics and Remote Systems, 1999.

[10] C.-F. Shu, A. Hampapur, M. Lu, L. Brown, J. Connell, A. Senior, Y. Tian, Ibm smart surveillance system (s3): a open and extensible framework for event based surveillance, in: Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance,, 2005, pp. 318–323.

[11] C. H. Lin, T. Lv, W. Wolf, I. B. Ozer, A Peer-to-Peer Architecture for Distributed Real-Time Gesture Recognition, in: Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on, Vol. 1, 2004, pp. 57–60 Vol.1.

[12] R. Dantu, S. P. Joglekar, Collaborative vision using networked sensors, in: Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on, Vol. 2, 2004, pp. 395–399 Vol.2.

[13] Q. Cai, J. K. Aggarwal, Tracking human motion in structured environments using a distributed-camera system, Pattern Analysis and Machine Intelligence, IEEE Transactions on 21 (11) (1999) 1241–1247.

[14] R. Collins, A. Lipton, H.Fujiyoshi, T. Kanade, Algorithms for cooperative multisensor surveillance, in: Proceedings of the IEEE, Vol. 89, 2001, pp. 1456–1477.

[15] J. Košecká, W. Zhang, Video Compass, in: Proceedings of the 7th European Conference on Computer Vision, Vol. 2353, 2002, p. 476.

[16] B. Bose, E. Grimson, Ground Plane Rectification by Tracking Moving Objects, in: Proceedings of the Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance (VS-PETS), 2003.

[17] S. Khan, M. Shah, Consistent labeling of tracked objects in multiple cameras with overlapping fields of view, Transactions on Pattern Analysis and Machine Intelligence 25 (10) (2003) 1355–1360.

[18] R. Pflugfelder, H. Bischof, Online Auto-Calibration in Man-Made Worlds, in: Digital Image Computing: Technqiues and Applications, 2005. DICTA ' 05. Proceedings, 2005, pp. 519–526.

[19] B. Rinner, M. Jovanovic, M. Quaritsch, Embedded Middleware on Distributed Smart Cameras, in: International Conference on Acoustics, Speech, and Signal Processing, No. 4, 2007, pp. 1381–1384.

[20] R. E. Schantz, D. C. Schmidt, Research advances in middleware for distributed systems, in: Proceedings of the IFIP 17th World Computer Congress - TC6 Stream on Communication Systems: The State of the Art, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2002, pp. 1–36.

[21] D. C. Schmidt, Middleware for Real-Time and Embedded Systems, Communications of the ACM 45 (6) (2002) 43–48.

[22] A. Pope, The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture, Addison Wesley, 1998.

[23] D. G. Schmidt, F. Kuhns, An overview of the real-time corba specification, Computer 33 (6) (2000) 56–63.

[24] Minimum CORBA Specification, `http://www.omg.org/technology/documents/formal/minimum_CORBA.htm`, last visited: Jun. 2008 (2008).

[25] D. C. Schmidt, D. L. Levine, S. Mungee, The design of the tao real-time object request broker, Computer Communications 21 (4).

[26] D. Box, Essential COM, Addison Wesley, 2007.

[27] R. Sessions, COM and DCOM: Microsoft's Vision for Distributed Objects, John Wiley & Sons, 1997.

[28] E. Pitt, K. McNiff, Java.rmi: The Remote Method Invocation Guide, Addison Wesley, 2001.

[29] Y. Yu, B. Krishnamachari, V. K. Prasanna, Issues in designing middleware for wireless sensor networks, Network, IEEE 18 (1) (2004) 15–21.

[30] M. M. Molla, S. I. Ahamed, A survey of middleware for sensor network and challenges, in: Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on, 2006, p. 6 pp.

[31] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler, Tinyos: An operating system for sensor networks, in: Ambient Intelligence, Springer Berlin / Heidelberg, 2005, pp. 115–148.

[32] C.-L. Fok, G. C. Roman, C. Lu, Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications, in: Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on, 2005, pp. 653–662.

[33] D. Georgoulas, K. Blow, Making Motes Intelligent: An Agent-Based Approach to Wireless Sensor Networks, WSEAS on Communications Journal 5 (3) (2006) 525–522.

[34] P. Bonnet, J. Gehrke, P. Seshadri, Towards Sensor Database Systems, in: Mobile Data Management, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2001, pp. 3–14.

[35] S. R. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, Tinydb: an acquisitional query processing system for sensor networks, ACM Trans. Database Syst. 30 (1) (2005) 122–173.

[36] M. Yokoo, S. Fujita, Trends of Internet auctions and agent-mediated Web commerce, New Gen. Comput. 19 (4) (2001) 369–388.

[37] T. Sandholm, eMediator A next generation electronic commerce server, Computational Intelligence 18 (4) (2002) 656–676.

[38] K. Stathis, O. de Bruijn, S. Macedo, Living memory: agent-based information management for connected local communities, Interacting with Computers 14 (6) (2002) 663–688.

[39] W.-S. E. Chen, C.-L. Hu, A mobile agent-based active network architecture for intelligent network control, Information Sciences 141 (1-2) (2002) 3–35.

[40] L.-D. Chou, K.-C. Shen, K.-C. Tang, C.-C. Kao, Implementation of Mobile-Agent-Based Network Management Systems for National Broadband Experimental Networks in Taiwan, in: Holonic and Multi-Agent Systems for Manufacturing, Springer Berlin / Heidelberg, 2003, pp. 280–289.

[41] M. Wooldridge, Intelligent Agents: The Key Concepts, Vol. 2322 of Lecture Notes in Computer Science, Springer-Verlag GmbH, 2002.

[42] Y. Aridor, M. Oshima, Infrastructure for mobile agents: Requirements and design, in: Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1998, pp. 38–49.

[43] MASIF Standard, `ftp://ftp.omg.org/pub/docs/orbos/98-03-09.pdf`, last visited: Jun. 2008 (1998).

[44] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, J. White, MASIF: The OMG Mobile Agent System Interoperability Facility, in: F. H. K. Rothermel (Ed.), Lecture Notes in Computer Scienc, Vol. 1477, Springer-Verlag GmbH, 1998, pp. 50–67.

[45] Foundation for Intelligent Physical Agents, Agent Communication Language, `http://www.fipa.org/repository/aclspecs.html`, last visited: Jun. 2008 (2007).

[46] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, D. Rus, D'Agents: applications and performance of a mobile-agent system, Softw. Pract. Exper. 32 (6) (2002) 543–573.

[47] C. Bäumer, T. Magedanz, Grasshopper — A Mobile Agent Platform for Active Telecommunication Networks, in: Intelligent Agents for Telecommunication Applications, Springer Berlin / Heidelberg, 1999, pp. 690–690.

[48] T. Wheeler, Voyager Architecture Best Practices, Tech. rep., Recursion Software, Inc. (March 2007).

[49] P. Marrow, M. Koubarakis, R.-H. van Lengen, F. J. Valverde-Albacete, E. Bonsma, J. Cid-Suerio, A. R. Figueiras-Vidal, A. Gallardo-Antolín, C. Hoile, T. Koutris, H. Y. Molina-Bulla, A. Navia-Vázquez, P. Raftopoulou, N. Skarmeas, C. Tryfonopoulos, F. Wang, C. Xiruhaki, Agents in Decentralised Information Ecosystems: the DIET Approach, in: Proceedings of the Artificial Intelligence and Simulation Behaviour Convention 2001 (AISB01), Symposium on Information Agents for Electronic Commerce, 2001, pp. 109–117.

[50] C. Hoile, F. Wang, E. Bonsma, P. Marrow, Core Specification and Experiments in DIET: A Decentralised Ecosystem-inspired Mobile Agent System, in: Proceedings 1st Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS2002), pp. 623-630, July 2002, Bologna, Italy, 2002, pp. 623–630.

[51] A. Fugetta, G. P. Picco, G. Vigna, Understanding Code Mobility, in: IEEE Transactions on Software Engineering, Vol. 24, IEEE Press, 1998, pp. 324–362.

[52] J. E. White, Telescript technology: mobile agents, Mobility: Processes, Computers, and Agents (1999) 460–493.

[53] A. J. N. Van Breemen, T. De Vries, An Agent based framework for designing Multi-controller Systems, in: J. Bradshaw, G. Arnold (Eds.), Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000), The Practical Application Company Ltd., Manchester, UK, 2000, pp. 219–236.

[54] N. R. Jennings, S. Bussmann, Agent-based control systems: Why are they suited to engineering complex systems?, Control Systems Magazine, IEEE 23 (3) (2003) 61–73.

[55] C. E. Pereira, L. Carro, Distributed real-time embedded systems: Recent advances, future trends and their impact on manufacturing plant control, Annual Reviews in Control 31 (1) (2007) 81–92.

[56] B. Chen, H. H. Cheng, J. Palen, Mobile-C: a mobile agent platform for mobile C-C++ agents, Softw. Pract. Exper. 36 (15) (2006) 1711–1733.

[57] H. H. Cheng, Ch: A C/C++ Interpreter for Script Computing, C/C++ User's Journal 24 (1) (2006) 6–12.

[58] D. C. Schmidt, An Architectural Overview of the ACE Framework, in: USENIX login, 1998.

[59] A. Doblander, A. Maier, B. Rinner, H. Schwabach, A Novel Software Framework for Power-Aware Service Reconfiguration in Multi-Reconfiguration in Distributed Embedded Smart Cameras, in: Proceedings of the 12th IEEE International Conference on Parallel and Distributed Systems (ICPADS'06), Minneapolis, Minnesota, USA, 2006, pp. 281–288.

[60] M. Quaritsch, M. Kreuzthaler, B. Rinner, H. Bischof, B. Strobl, Autonomous Multi-Camera Tracking on Embedded Smart Cameras, EURASIP Journal on Embedded Systems 2007 (2007) 10.

[61] M. Bramberger, M. Quaritsch, T. Winkler, B. Rinner, H. Schwabach, Integrating Multi-Camera Tracking into a Dynamic Task Allocation System for Smart Cameras, in: Proceedings of the IEEE International Conference on Advanced Video and Signal Based Surveillance, 2005.

[62] J. Llinas, D. L. Hall, An Introduction to Multi-Sensor Data Fusion, in: Proceedings of the 1998 IEEE International Symposium on Circuits and Systems, Vol. 6, 1998, pp. 537–540.

[63] D. L. Hall, J. Llinas, Handbook of Multisensor Data Fusion, CRC Press, 2001.

[64] B. V. Dasarathy, Information Fusion - what, where, why, when, and how?, in: Proceedings of the 4th International Conference on Information Fusion, Vol. 2, 2001, pp. 75–76.

[65] B. Moshiri, M. R. Asharif, R. H. Nezhad, Pseudo Information Measure: A New Concept for extension of Bayesian Fusion in Robotic Map Building, Journal of Information Fusion 3 (2002) 51–68.

[66] V. Vapnik, Statistical Learning Theory, Wiley, New York, US, 1998.

[67] A. Klausner, A. Tengg, B. Rinner, Distributed multi-level Data Fusion for Networked Embedded Systems, IEEE Journal on Selected Topics in Signal Processing 2 (4) (2008) 536–555.

[68] A. Doblander, A. Zoufal, B. Rinner, A novel software framwork for embedded multiprocessor smart cameras, ACM Transactions on Embedded Computing Systems. Accepted for publication.