IMPROVED AGENT-ORIENTED MIDDLEWARE FOR DISTRIBUTED SMART CAMERAS

Markus Quaritsch[1], *Bernhard Rinner*[2], *Bernhard Strobl*[3]

[1]Graz University of Technology, Institute for Technical Informatics 8010 Graz, AUSTRIA
[2]Klagenfurt University, Institute of Networked and Embedded Systems, 9020 Klagenfurt, AUSTRIA
[3]Austrian Research Centers GmbH, Smart Systems Division, 1020 Wien, AUSTRIA

ABSTRACT

In the recent past, much effort has been put into the development of distributed vision systems with smart cameras as key components. Smart cameras combine video sensing, processing and communication within a single embedded device and provide sufficient on-board infrastructure to carry out high-level video analysis tasks. Networks of smart cameras help to overcome some hard problems inherent to single-camera systems by providing multiple views of a scene.

This paper reports on an improved, agent-oriented middleware for embedded smart cameras. Each image processing task is represented by an agent resident on a smart camera within the network. Agents are able to move from one camera to another as needed during run-time. An agent is comprised of the high-level application logic and the image processing algorithm which is executed on the processing unit.

The presented middleware is also designed for distributed image processing where two or more cameras can cooperate for a single task. In the paper we discuss the requirements for such an agent-oriented middleware capable of supporting distributed image processing. Further, we describe the architecture of our middleware implementation. The evaluation of our current middleware implementation shows significant performance improvements compared to our previous Java-based implementation.

Index Terms— smart camera, embedded system, middleware, distributed image processing

1. INTRODUCTION

Smart cameras have gained increasing research interest in the last few years. This trend is driven by recent advances in the field of image processing and efforts on integrating image processing in embedded systems. Smart cameras combine video sensing, processing and communication within a single embedded device [1, 2].

Recently, much effort has also been put into the development of distributed vision systems with smart cameras as key components. Networks of distributed smart cameras [3] are an emerging technology for a broad range of important applications including smart rooms, surveillance, 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. The target applications for distributed smart camera systems pose strong requirements on the camera's hardware and software. Typically, the cameras have to execute demanding video processing and compression algorithms.

Developing distributed image processing applications is challenging and implementing these on a network of smart cameras is also difficult. A substantial system-level software, therefore, would strongly support the implementation. Unfortunately, embedded platforms with limited resources typically do not provide middleware services well-known on general-purpose platforms. In [4] we have discussed a set of important services for a smart camera network and exemplified the benefits of these services on a concrete application. Indeed, we could demonstrate the benefits of such a middleware but the achieved performance was very poor. This is a consequence of implementing the middleware in Java which, unfortunately, causes a considerable performance gap on embedded systems due to the lack of a just-in-time compiler.

The agent-oriented programming paradigm is rather uncommon for embedded systems up to now. This is mainly because of the undesirable non-deterministic behavior of agent systems. However, smart camera networks can benefit from harnessing the agent-oriented approach. A smart camera network typically has to carry out several different and independent tasks. These tasks can be mapped to agents, whereas a task is fulfilled by one or more agents. Each agent focuses on its main job and cooperates with other agents in order to fulfill its mission. Starting an additional task in a smart camera network, thus, is as simple as creating a new agent in the network.

In this paper, we focus on the implementation of an efficient, lightweight distribution service for embedded smart cameras. Our proposed middleware follows the agent-oriented programming paradigm, using agents as abstraction of image processing tasks. Compared to our previous work [4], we have significantly improved the performance of the distribution layer concerning network communication, interaction with image processing tasks, as well as resource utilization.

1-4244-1354-0/07/$25.00

The implementation also incorporates the efficient transmission of raw-images which is important for collaborative image processing.

The remainder of this paper is organized as follows. Section 2 first gives a short survey of current smart camera architectures and identifies similarities among those. Afterwards, related middleware approaches are presented. Section 3 identifies the potential of distributed smart cameras. Section 4 is dedicated to our proposed smart camera middleware. We first identify the requirements for such a middleware and sketch the basic architecture. Then we focus on the distribution layer which is the central part of the middleware. Section 5 demonstrates the efficiency of our implementation compared to our Java-based middleware. Section 6 concludes this paper with a brief discussion.

2. BACKGROUND AND RELATED WORK

2.1. Smart Camera Architectures

Image processing in general is a very computing intensive task due to the huge amount of data to be processed. Depending on the level of abstraction, processing requirements increase dramatically. Hence, implementing image processing algorithms on embedded smart cameras is very challenging. Various architectures have been introduced, focusing on a concrete set of image processing algorithms while minimizing the resource requirements, especially energy consumption.

Kleihorst et al. present in [5] a smart camera mote optimized for very low power consumption. The smart camera mote consists of basically four components, one or two VGA resolution color image sensors, an SIMD processor for low-level image processing, a general purpose processor for high-level processing and control, and a communication module. Both processors are coupled using a dual-port RAM which enables them to work in a shared workspace on their own processing space. The SIMD processor allows to process 320 pixels of a line at once. The host controller is a 8051 based ATMEL processor.

Fleck and Straßer use in their work [6] a commercially available smart camera for embedded image processing. This camera is comprised of a single CCD sensor with VGA resolution, a Xilinx FPGA for low-level image processing, and a Motorola PowerPC processor as host controller. The host controller is operated by a Linux kernel optimized for embedded systems.

Bramberger et al. propose a heterogeneous multi-processor smart camera [2]. The architecture of this smart camera contains three main units: a CMOS image sensor which delivers color images up to VGA resolution, the processing unit comprised of one or more digital signal processors (DSPs), and an ARM based network processor for overall system coordination and communication. The operating system used on this camera is also Linux.

These examples of smart camera architectures aim on different types of image processing, ranging from low-level pixel filtering to high-level motion detection and object tracking, according to the available computing power and energy consumption. However, all of these approaches have a similar architecture: A dedicated processor or DSP is used for image processing while a general purpose processor is responsible for camera coordination and communication. This trend is also obvious in other commercially available smart cameras. The IP-Camera advertised by Nuvation [7] for example uses a DaVinci DM6446 processor, which combines a DSP core and an ARM core in a single chip. This camera is operated by a Linux kernel as well.

The emerging trend of using Linux as operating system for embedded devices is also evident in the research area of smart cameras. Linux brings a whole set of features usually available on general-purpose PCs to embedded systems and supports a wide variety of platforms. Due to its open source license it can be easily adapted to new platforms and tuned for special needs.

2.2. Related Middleware Approaches

On general-purpose platforms distributed applications are often implemented on the basis of a reusable middleware system which encapsulates networking and data transfer [8]. Prominent representatives are DCOM and CORBA, for example. These component-based middleware systems are targeted for general-purpose computing and are not suitable for resource limited devices. There exist also lightweight variations of traditional CORBA services for resource constrained real-time systems, e.g., by the Real-Time CORBA (RT-CORBA) specification and its "TAO" implementation [8]. However, this approach is still very resource consuming.

Middleware systems based on the Java programming language are also very common on general-purpose platforms. Java RMI (Remote Method Invocation) and EJB (Enterprise Java Beans) are well-known examples for this kind of middleware. Agent-based systems like Voyager [9], Grasshopper [10] or Diet-Agents [11] are also used as middleware. The agent-system not only abstracts the hardware platform and distributed aspect but also introduces the agent-based programming paradigm. Platform independence is inherent to all these approaches as the actual hardware is hidden by the Java virtual machine. However, using Java on embedded systems is not an efficient choice because it is heavy weighted in terms of memory footprint and computation.

Another research area related to smart camera networks are wireless sensor networks (WSNs) [12]. Distributed and collaborative signal processing is inherent to WSNs. In the recent past, there is ongoing research on middleware systems for WSNs. However, middleware approaches designed for WSNs are not applicable to smart camera networks due to

different design aspects and resource constraints [13, 14]. For WSNs, the focus is on reliable services in ad-hoc networks and energy awareness. The amount of data to be processed and the available communication bandwidth is also much smaller in WSNs.

3. SMART CAMERA NETWORKS

Networks of distributed smart cameras (DSCs) are well suited for distributed image processing. Although distributed image processing introduces several difficulties, we believe that the problems which can be addressed by this approach are more important than the challenges of designing and implementing networks of smart cameras. The main benefit of DSCs compared to single-camera systems is that a network of smart cameras may provide multiple views of a scene from different views.

Occlusion is a major problem for single cameras. In a multi-camera setup it is more likely that multiple views of an object are available. Hence, parts of an object which are occluded in one field of view may be visible from another view.

Traditional multi-camera setups usually use a central node which processes the different views of a scene. Obviously, the scalability of such centralized systems is limited as the number of cameras directly influences the required communication bandwidth and processing power. Local image processing on the smart cameras significantly improves the scalability of the overall system and also reduces the required communication bandwidth considerably. Instead of sending the raw image data each camera processes the acquired images locally and communicates abstracted information of a scene. Nevertheless, for some applications it may be necessary to process the images of different views jointly and thus transmit raw image data. In contrast to centralized methods, a decentralized approach shows better scalability because communication is done between typically two or three collaborating cameras.

A network of smart cameras with multiple views of a scene may also overcome failures of individual cameras. Distributed computing enables fault-tolerance and helps to increase the reliability of the multi-camera system.

4. MIDDLEWARE FOR DISTRIBUTED SMART CAMERAS

Designing and implementing software for distributed systems is rather challenging. One has to cope with concurrency issues, unreliable and basically non-deterministic network connections between the hosts, as well as platform dependencies. These issues are not specific to DSCs but apply for distributed computing in general. However, the requirements for a middleware for distributed image processing on embedded devices are significantly different.

Applications of smart cameras for distributed image processing, therefore, either re-implement network communication itself (e. g. [15]) or adapt general-purpose middleware (e. g. [16]) causing a dramatic performance penalty.

4.1. Middleware Requirements

A middleware for image processing on embedded smart cameras has to provide basic functionality of general-purpose middleware like abstracting network communication and providing mechanisms to interact with other hosts but also has to offer services specific for distributed image processing. In the following, we describe important services and properties of a middleware for DSCs.

Lightweight. For a DSC middleware it is essential to be lightweight. As the computing power on embedded systems is limited, the overhead introduced by a middleware should be minimal. The memory requirements also have to be considered because this is a limited resource too.

Abstraction of image processing. A middleware for DSCs has to support the abstraction of image processing and encourage the separation of DSC applications into application logic and image processing algorithm. The application logic on the one hand contains the high-level logic for performing a certain task, e.g. generate an event if there is motion in a scene or collaborate with other cameras in order to track an object. The image processing algorithm on the other hand does the low-level pixel processing and extracts features of the acquired images, for example detect motion in a sequence of images or classify objects in a scene.

The separation of application logic and image processing makes the image processing exchangeable. Even for the simple case of generating an event when motion is detected in a scene, different algorithms may be used, depending on the actual requirements (e.g. different background models). Thus, the same application logic may be configured to use different image processing algorithms depending on the current needs.

Collaborative image processing. Collaborative, distributed image processing is the most important and challenging aspect of DSC networks. There is a whole class of image analysis challenges which require or can benefit of having multiple views. For example, when performing face recognition it may be helpful to have multiple views of the person of interest from different perspectives either to deal with occlusion or increase reliability. Collaborative image processing can be done in two ways. Either, each camera performs the image processing locally and only abstracted information is exchanged among collaborating cameras. This case does not introduce additional requirements for the middleware as message oriented communication between individual nodes is a basic feature of each middleware. The second, and more challenging,

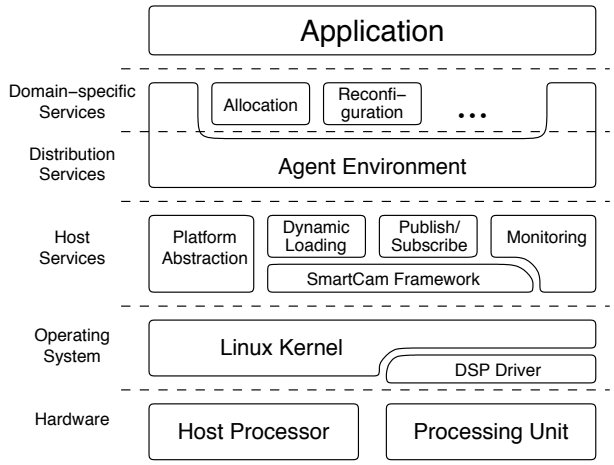

Figure 1: The architecture of a middleware for DSC networks.

approach is that the image processing task is executed only on a single camera incorporating the images of other cameras. This imposes, that the raw images acquired by one camera have to be transmitted to another camera. A middleware for DSC networks thus has to provide a service which gives other cameras access to the raw image data—either of the complete scene or only a certain region of interest. Of course, the available resources, especially bandwidth, have to be taken into account.

Synchronization. The collaboration of cameras for a certain task implicates to have a well-defined temporal relation among the individual cameras in the DSC network. Otherwise it is impossible to bring images acquired by different cameras in a common context. The required temporal accuracy depends on the image processing algorithm and may range from frame synchronization to a very relaxed synchronization.

4.2. Proposed Middleware Architecture

The individual components of our proposed middleware for DSC networks are organized in a layered architecture, whereas each layer abstracts the services provided by the underlying layer. The functionality of each layer is adapted from [8]. Figure 1 sketches our proposed architecture. This architecture is similar to the one presented in [4] but has been adapted to our lightweight middleware approach. Although some modules—especially in the host services layer—are specific to the used smart camera hardware, this approach is also suitable for other smart camera platforms. Only the platform related modules such as the *SmartCam Framework* and the *DSP Driver* have to be adapted for new platforms.

Operating system. The host processor is operated by a standard Linux kernel optimized for embedded systems. A custom kernel module (*DSP Driver*) handles the communication between the DSPs as well as the DSPs and the host processor. This layer is mostly independent of the smart camera platform as long as the Linux kernel supports the host processor. Only the communication between the individual processors depends on the concrete platform.

Host services. The foundation of the host services layer is the *SmartCam Framework*. This module provides a message-oriented communication mechanism between applications on the host processor and the image processing tasks on the DSPs. The services for (1) *dynamic loading*, (2) *publish/subscribe* [17], and (3) *monitoring* operate on top of the SmartCam Framework. The dynamic loading service allows an application to load and unload image processing algorithms to the DSPs dynamically during run-time. The publish/subscribe service offers a flexible communication mechanism between data sources and data sinks whereas the data sources and data sinks can be located on different processors. The monitoring service provides information about the resource utilization of the various hardware components.

The *Platform Abstraction* module encapsulates native low-level operating system functionality such as network communication and concurrency mechanisms. For this module, the *ADAPTIVE Component Environment* (`http://www.cs.wustl.edu/~schmidt/ACE.html`) is used. It supports a great number of operating system and provides an object-oriented API.

Adapting this layer to a different smart camera platform basically requires to replace the SmartCam Framework and provide a mechanism for sending messages between the host processor and the processing unit.

Distribution layer. The distribution layer is the main component of the DSC middleware. While the lower layers provide services for applications on a single camera, this layer integrates multiple smart cameras to a distributed image processing system. Thus, the basic service provided by this layer is remote communication, which can either be message oriented or stream oriented. Message oriented communication is typically used for exchanging control messages and data while stream oriented transmission is intended for sending raw-images from one camera to another or stream encoded video. Details on this layer are given in section 4.3.

This layer is independent of the smart camera architecture as long as the services listed in the *host services* layer are available.

Domain-specific services. On top of the distribution layer reside the domain-specific services. These are services which are on the one hand very specific to the application domain (e.g. video surveillance) and on the other hand common for applications of a DSC network. Such services are, for ex-

ample, camera calibration and localization, reconfiguration or QoS management.

4.3. Agent-Oriented Distribution Layer

The distribution layer is the central part of the DSC middleware. It is responsible for handling the communication between cameras and provides a foundation for the applications in a DSC network. Therefore, it is crucial to implement this layer in an efficient manner, keeping the overhead minimal.

We have chosen to implement applications using the agent-oriented programming paradigm. This form of abstraction has already proven to be suitable for image processing tasks (c.f. [16]). The distribution layer, therefore, can also be seen as an agency providing the environment for agents which form the application layer and domain-specific services layer.

Each agent represents a task in the DSC network. This is, for example, generating an event in the case of motion in a scene or tracking a person. The agent itself contains the application logic and has attached the image processing algorithm, which is executed on the processing unit. The agent loads the image processing algorithm on the processing unit and uses the information supplied by the image processing task to take further actions, e.g. trigger another agent. This approach also encourages a strict separation between application logic and image processing.

Each camera hosts an agency which provides the environment for the agents. An agency allows agents to create new agents of a certain type, communicate with other agents on the same camera or another camera, and interact with the processing unit on the local camera in order to do image processing. While agent creation and agent communication are treated as a basic features, these are provided by the agency itself. Further services, like interaction with the processing unit, are provided by additional agents available on each camera.

Allowing agents to migrate from one camera to another is fundamental for autonomous, self-adapting behavior of DSC networks. Unfortunately, code migration is not supported by C++. An agent can, however, migrate to another camera by remotely creating a new agent and initializing it accordingly. Afterwards, the original agent can terminate. If the agent class is not known on the new camera, a dynamic library can be sent to the camera which provides the agent definition.

The distribution layer further has to provide a mechanism which allows to distribute the raw-images among collaborating cameras. For this high-bandwidth stream oriented communication we propose to use the Stream Control Transmission Protocol (SCTP) [18] which has been standardized recently. SCTP is a connection-oriented point-to-point transport layer protocol (OSI layer 4) providing a reliable delivery over an IP network. It combines the benefits of TCP and UDP while cutting their drawbacks and introduces a set of new features valuable for DSC networks. A selection of features is described in the following.

Multiple streams in a single association. SCTP uses the term association for what is called connection in the TCP protocol for various reasons. An SCTP stream is a unidirectional logical data flow within a SCTP association. An arbitrary number of these logical streams can be used per SCTP association in both directions. For each stream, the data order is preserved. This novel feature is the basis for a set of other features.

Using multiple streams avoids the head-of-line blocking which may occur when a TCP receiver is forced to re-sequence packets that arrive out of order because of network reordering or packet loss. In SCTP, if data on a stream is lost, only this stream is blocked waiting for retransmission while all other streams are not affected.

Multiple delivery modes. SCTP offers multiple delivery modes. Within a stream, messages can be delivered in strict order-of-transmission, like TCP, partially ordered, or unordered, like UDP. Additionally, SCTP not only supports reliable transmission but also unreliable transmission.

Multihoming. Multihoming is an attempt to increase the network resilience to failed interfaces. If the host is equipped with multiple network interfaces, SCTP may use one or more of these interfaces for a single association. In the case of a failure in the network, SCTP automatically switches to another interface preserving the established association. This is transparent to the application.

TCP-friendly congestion control. As SCTP is used together with TCP and UDP in the same network, it is important that SCTP uses a TCP-friendly congestion control. This means, that the available bandwidth is shared fairly between SCTP and TCP. UDP, in contrast, has no congestion control mechanism. Hence, high bandwidth UDP transmissions (e.g. streaming video data, or exchanging raw image data) lead to an unfair sharing of bandwidth with other protocols like TCP or SCTP.

Taking these features into account, the SCTP protocol is well suited DSC networks due to its rich set of features and high configurability. For collaborative image processing, individual regions of interest (ROIs) can be transmitted via separate streams between cameras. This supports independent processing of each ROI at the receiver, especially in the case of packet loss. Encoded video data may be transmitted via multiple streams of different reliability. For example, I-frames are transmitted partially reliable while P-frames are transmitted unreliable.

When using UDP for streaming the raw-images, a major drawback would be the lack of a congestion control mecha-

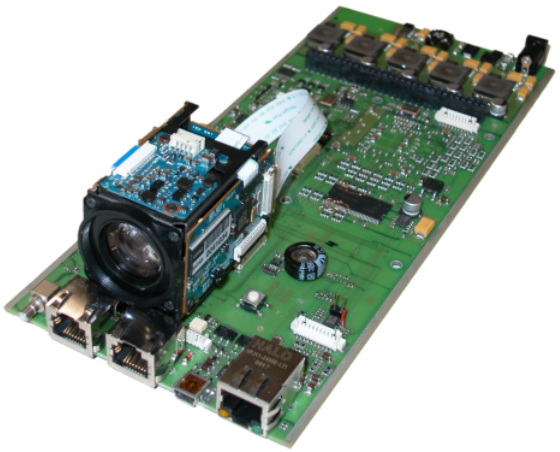

Figure 2: Smart Camera.

Agency	1,8 MB
Java Classpath	11,0 MB
Java VM	0,5 MB
Total:	13,3 MB

(a) Java Implementation

Agency	0,8 MB
Libraries	2,1 MB
Total:	2,9 MB

(b) C++ Implementation

Table 1: Comparison of the code size

nism, while using TCP may lead to increased delays caused by its reliability and the potential for head-of-line blocking.

5. EXPERIMENTAL RESULTS

We have implemented our proposed agent-oriented lightweight DSC middleware in C++. The evaluation first compares the C++ implementation with our previous Java-based middleware [16] and discusses the achieved performance. We then analyze whether the SCTP protocol is suitable for transmitting raw images from one camera to another in order to do collaborative image processing.

5.1. SmartCam Architecture

For the evaluation we have used our SmartCamera [2] which is comprised of an ARM-based host processor and two DSPs from Texas Instruments. The processors are connected via a PCI bus. The operating system of the host processor is a standard Linux kernel version 2.6.17. Figure 2 depicts our smart camera.

5.2. Agency Implementation

Embedded devices typically have tight resource constraints, especially with respect to computing power and memory. Therefore, a lightweight middleware has to use the scarce resources efficiently. Table 1 compares the code sizes of the C++ implementation with the Java-based Implementation. The major benefit of Java, its comprehensive class library, has the downside of requiring plenty of memory. Thus, the Java-based implementation is with 13,3 MB more than four times larger than the C++ implementation.

	C++	*Java*
Loading dynamic executable	8 ms	180 ms
Initializing tracking algorithm (5 frames @ 20 fps)	250 ms	250 ms
Creating agent on next camera	18 ms	2130 ms
Reinitialize tracking algorithm on next camera	2 ms	40 ms

Table 2: Comparison of the improved C++ middleware implementation and Java implementation

5.3. Loading Image Processing Tasks

The image processing tasks (i.e. the CamShift tracker) are loaded and unloaded dynamically by the agent as needed. Therefore, we have evaluated the time required for loading a dynamic executable to the DSP as well as initializing the image processing task.

Table 2 shows the time intervals for loading, initializing and reinitializing the CamShift tracking algorithm opposed to the results of our Java based middleware implementation.

Considering the obtained results in table 2, it can be seen, that the improved implementation of our middleware is about 20 times faster than the Java implementation. Loading a dynamic executable takes no more than 8 ms and re-initializing the tracking algorithm requires about 2 ms. The time intervals required for initializing the tracking algorithm are equal because this is done by the tracking algorithm itself and requires no interaction with the agent.

Creating an agent on the next camera took more than 2 seconds using the Java implementation. this time interval could be decreased significantly to less than 20 ms.

5.4. Transmitting RAW-Images

This section focuses on collaborative image processing. Therefore, we compare the SCTP protocol and the TCP protocol for streaming raw image data from one camera to another. We transmitted a ROI of various sizes over a 100 MBit wired Ethernet network under different packet-loss conditions. When using SCTP, we used either a single stream or four concurrent streams with reliable transmission. In the case of four streams the ROI is split up into four equally sized chunks—one for each stream. For both protocols we used the standard configuration of the Linux kernel. Table 3 shows the average transmission time of a ROI over 200 transmissions.

ROI size	TCP	SCTP	
		(1 stream)	(4 streams)
100×100	2 ms	2 ms	3 ms
200×200	5 ms	5 ms	7 ms
352×288	12 ms	9 ms	17 ms

(a) 0% packet loss

ROI size	TCP	SCTP	
		(1 stream)	(4 streams)
100×100	7 ms	2 ms	2 ms
200×200	12 ms	2 ms	4 ms
352×288	24 ms	30 ms	8 ms

(b) 1% packet loss

ROI size	TCP	SCTP	
		(1 stream)	(4 streams)
100×100	19 ms	2 ms	2 ms
200×200	32 ms	2 ms	4 ms
352×288	35 ms	156 ms	10 ms

(c) 5% packet loss

ROI size	TCP	SCTP	
		(1 stream)	(4 streams)
100×100	32 ms	4 ms	3 ms
200×200	46 ms	11 ms	6 ms
352×288	46 ms	397 ms	11 ms

(d) 10% packet loss

Table 3: Comparison of TCP and SCTP for streaming raw image data for different packet loss conditions.

In the case of no packet loss (c.f. Table 3a), SCTP is comparable to TCP for small ROIs when using a single stream and slightly slower when using four streams. This is caused by the rather small chunks for each stream and the associated transmission overhead. The transmission time increases for all three protocols according to the ROI size. However, SCTP with only one stream performs better than TCP.

In the case of packet loss (c.f. Table 3b–d), the transmission time for TCP increases according to the packet loss rate as well as the ROI size (up to 35 ms for transmitting a ROI of CIF resolution for 5% packet loss). In contrast to this, the average transmission time for SCTP is almost independent of the packet loss rate for small ROIs. For larger ROIs, TCP shows better performance than SCTP when using only one stream. But using four streams significantly reduces the transmission times for SCTP and thus outperforms TCP. A ROI of CIF resolution is on average transmitted in about 10 ms when using four SCTP streams although the packet loss rate is 10%. This illustrates the advantage of preventing head-of-line blocking in SCTP.

The obtained results show that SCTP performs better when transmitting large ROIs by using multiple streams. In the case of small ROIs, SCTP is comparable to TCP. Thus, SCTP is a good choice for transmitting raw image data. However, it is beneficial to know the size of the ROI in order to adapt the number of streams of the SCTP association accordingly.

6. CONCLUSION

In this paper we have presented an improved middleware for networks of distributed smart cameras. We focus on the distribution layer of the middleware which follows the agent-oriented programming paradigm. Image processing tasks in the smart camera network are abstracted by agents. Each agent contains the high-level application logic, e.g. generate an alarm when there is motion in a scene, and has attached the image processing algorithm which does the actual pixel processing, e.g. detect motion in a scene. First evaluation results show a significant performance increase compared to our previous Java-based implementation.

Our proposed middleware is further designed for collaborative image processing which requires a camera to access the raw image data of another camera. Therefore we propose the use of SCTP, a recently standardized transport layer protocol. Due to the various new features, SCTP is well suited for transmitting small as well as large image chunks. Using multiple streams also avoids head-of-line blocking which reduces the transmission time significantly in the case of packet loss.

7. REFERENCES

[1] Wayne Wolf, Burak Ozer, and Tiehan Lv, "Smart cameras as embedded systems," *Computer*, vol. 35, no. 9, pp. 48–53, Sept. 2002.

[2] Michael Bramberger, Andreas Doblander, Arnold Maier, Bernhard Rinner, and Helmut Schwabach, "Distributed Embedded Smart Cameras for Surveillance Applications," *Computer*, vol. 39, no. 2, pp. 68 – 75, 2006.

[3] Bernhard Rinner and Wayne Wolf, Eds., *Proceedings of the Workshop of Distributed Smart Cameras*, Boulder, CO, USA, October 2006.

[4] Bernhard Rinner, Milan Jovanovic, and Markus Quaritsch, "Embedded Middleware on Distributed Smart Cameras," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 2007, vol. 4, pp. 1381–1384.

[5] Richard Kleihorst, Ben Schueler, Alexander Danilin, and Marc Heijligers, "Smart Camera Mote with High Performance Vision System," in *Proceedings of the Workshop on Distributed Smart Cameras (DSC-06)*, Boulder, CO, USA, October 2006, pp. 17–21.

[6] Sven Fleck and Wolfgang Straßer, "Adaptive Probabilistic Tracking Embedded in a Smart Camera," in *Proceedings of IEEE Embedded Computer Vision Workshop (ECVW) in conjunction with IEEE CVPR 2005*, 2005, pp. 134 – 134.

[7] Nuvation, "Pixim IP Camera," http://www.nuvation.com/ip/ipcamera.html.

[8] Douglas C. Schmidt, "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, no. 6, pp. 43–48, June 2002.

[9] Thomas Wheeler, "Voyager Architecture Best Practices," Tech. Rep., Recursion Software, March 2005.

[10] C. Bäumer, M. Breugst, S.Choy, and T.Magedanz, "Grasshopper – A Universal Agent Platform based on OMG MASIF and FIPA Standards," in *First International Workshop on Mobile Agents for Telecommunication Applications (MATA'99)*, October 1999, pp. 1–18.

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

[12] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, March 2002.

[13] Mohammad M. Molla and Sheikh Iqbal Ahamed, "A Survey of Middleware for Sensor Networks and Challenges," in *Proceedings of the 2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, Columbus, Ohio, USA, Aug 2006, pp. 223–228, IEEE.

[14] Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna, "Issues in Designing Middleware for Wireless Sensor Networks," *Network, IEEE*, vol. 18, no. 1, pp. 15–21, Jan/Feb 2004.

[15] Sven Fleck, Florian Busch, Peter Biber, and Wolfgang Straßer, "3D Surveillance – A Distributed Network of Smart Cameras for Real-Time Tracking and its Visualization in 3D," in *Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*, Jun. 2006, pp. 118 – 126.

[16] Markus Quaritsch, Markus Kreuzthaler, Bernhard Rinner, Horst Bischof, and Bernhard Strobl, "Autonomous Multi-Camera Tracking on Embedded Smart Cameras," *EURASIP Journal on Embedded Systems*, vol. 2007, 2007.

[17] Andreas Doblander, Bernhard Rinner, Norbert Trenkwalder, and Andreas Zoufal, "A Middleware Framework for Dynamic Reconfiguration and Component Composition in Embedded Smart Cameras," *WSEAS Transactions on Computers*, vol. 5, no. 3, pp. 574–581, March 2006.

[18] R. Stewart and C. Metz, "SCTP: new transport protocol for TCP/IP," *IEEE Internet Computing*, vol. 5, no. 6, pp. 64–69, 2001.