Flexible Clustering in Networks of Smart Cameras

Bernhard Dieber, Bernhard Rinner
Institute of Networked and Embedded Systems
Klagenfurt University, Austria
{Bernhard.Dieber, Bernhard.Rinner}@uni-klu.ac.at

Nikolaus Viertl
Video and Security Technology
Austrian Institute of Technology, Vienna
Nikolaus.Viertl@ait.ac.at

Abstract

Advances in sensors, networking and embedded computing have driven a paradigm shift in the control of camera networks. The processing of the image data has migrated from central servers to the camera network following fully distributed or clustered approach. In clustering, the data is analyzed within a group of cameras, and the abstracted data is then forwarded to other clusters.

In this paper we present a flexible and scalable software tool that supports clustering in smart camera networks. This tool supports the dataflow-oriented processing within a group of cameras. A plug–in-mechanism facilitates flexibility and scalability; data processing can be migrated between the cluster node and the smart cameras. We demonstrate the feasibility of our tool in a multi-camera person tracking case study.

1. Introduction

A huge number of camera networks have been deployed so far. Most of these networks have become larger, more ubiquitous and more embedded over the last recent years. Although the number of deployments has risen dramatically, control and coordination of these networks is still a challenging problem. Typical control and coordination tasks include camera selection, camera hand-off and fusion.

Fundamentally, several control strategies are possible in camera networks. In the *centralized approach*, a dedicated node has access to all required data in order to control the cameras. In this setting, the cameras typically stream their (raw) data to the central node where control and computation takes place. In the *distributed approach*, the control is not assigned to a single node but to a set of cooperating node (see for example [3, 5, 8]). Here, the assignment of control tasks changes dynamically and is often managed in an ad-hoc and peer-to-peer manner. In the *clustered approach*, the (raw) data is processed within a group of cameras (cluster), and abstracted data is transferred among different clusters (examples can be found in [1, 2, 6]). This imposes a hierarchical control strategy in the network. Clustering is a very natural approach for many multi-camera applications because (i) the network is often hierarchically structured, (ii) the monitored activities in the network have a strong locality and (iii) it offers high scalability and flexibility.

This paper focuses on a framework for developing and evaluating cluster-based multi-camera applications. Our framework manages the data flow within a cluster of cameras, i.e., it reads data from a variable number of n inputs (cameras), processes the input streams and generates a variable number of m outputs (compare Figure 1). Great effort has been put into flexibility and composability of our framework to support multi-camera application development. The plug-in mechanism of our framework enables an easy adaptation of number and type of inputs (and outputs). Various software modules are automatically composed into the overall processing pipeline. A monitoring utility checks important performance parameters to support the evaluation of the application.

Since input and output as well as the processing pipeline can be easily adapted, the framework supports processing of raw, compressed or abstracted data within the camera cluster. It is therefore well suited for traditional as well as smart camera networks [7]. We demonstrate this high flexibility by two case studies: person detection in a traditional multi-camera network and person detection in a network of embedded smart cameras.

The remainder of this paper is organized as follows: Section 2 discusses the research questions of this work. In Section 3 we present the requirements and the design of our software framework. Section 4 describes a case study and experimental results. We conclude this paper with a brief discussion of future work.

2. Research Questions

Cameras can be clustered *e.g.* by their field of view or physical location. The network topology itself can also be a clustering criterion.

For each application scenario a different clustering method is optimal. Therefore it is necessary to evaluate and

compare different clusterings. We focus (but do not limit) our research on scalability in clustered networks.

2.1. Scalability

Clusters can be scalable with respect to different aspects which are closely related to each other, *e.g*. the number of cameras in the cluster, the processing workload in the cluster nodes or the inter– and intra–cluster data flow.

We want to evaluate the trade off among the following–mututaly influencing–scalability factors.

Number of inputs: The number of cameras connected to each cluster node (CN) can be varied. A larger number of cameras in the cluster results in an increased data volume and hence increased processing load in the CN. Therefore software in the CN must be scalable regarding the number of inputs.

Data flow: The input data for each cluster node can can be chosen differently in terms of data format or data type. Examples are raw image data or higher abstracted information (like scene descriptions).

Processing pipeline: Depending on input type and number of input sources, a CN can perform different types of fusion with a variable number of processing steps (see also [2]). Consequently, this also influences the type of output in each cluster node. This output is higher abstracted than the single inputs (e.g. for raw image data as input a cluster node may deliver a scene description with all object found in all views).

Number of outputs: A CN can have more than one output (see for example [3]). Possible output channels are the transfer of results to a superordinate node, persistent storage, event dispatching or visualization for human supervisors.

2.2. Practical Experiments and Case Studies

We want to practically evaluate certain clustering methods in real networks which brings up the need to record performance indicators at every node in the network. This creates huge amounts of data, thus a mechanism to collect and analyze performance data must be provided.

For large networks, applying clustering is a non-trivial task because software must be deployed to all nodes. The software configuration may be similar but is not the same for every CN. Depending on the data flowing in and out of CNs the actual processing in the nodes changes. In order to efficiently evaluate different clusterings a mechanism for mass-deployment of software to all CNs in the network must be provided.

2.3. Multi-cluster Modeling

In order to perform practical evaluation in a clustered camera network, the network must be modeled and constructed. We model clusters and cluster hierarchies with focus on the intra– and inter–cluster data flow, i.e. a CN processes multiple data streams from smart cameras and potentially transfers results to nodes in higher levels.

3. Framework

To facilitate implementation and evaluation of clustered camera networks we present a software platform that runs on every cluster node. In this section we first explain the requirements that we identified for this software. We then present our software design which fulfills those requirements and explain further details on the framework.

3.1. Requirements

For our software we identified some key requirements that have to be fulfilled in order to provide a platform that can be used in large–scale clustered networks.

Scalability: The software has to be able to accept a variable number of inputs and deliver a variable number of outputs. This is essential to evaluate clustering by varying the number of connected cameras. Additionally, the processing work performed in an instance must be variable and easy to change. To allow different types of work to be performed in the cluster node, a mechanism to easily exchange the processing part must be provided.

Composability and Flexibility: We want applications which are built on top of our software to be composed from exchangeable modules. This increases the level of code reuse and thus speeds up application development.

To support a variety of application scenarios, the software should not restrict the data flow by means of data type or data content.

Deployment Support and Monitoring: The software is intended to provide facilities to easily get a cluster node up and running. This includes selecting and composing the processing part, collecting the necessary configuration values and deploying the software to all cluster nodes. Since cluster nodes on the same hierarchy level may have similar configurations a mechanism to facilitate the creation of multiple similar software instances is needed.

To collect performance indicators in a large network, a cluster node must be able to monitor itself and collect relevant data. In different clustering scenarios different performance data is of relevance. Thus it must be possible to select those performance parameters to be monitored. In many cases it is not sufficient to just monitor process specific parameters like CPU usage or memory consumptions but it will be necessary to analyze the data flowing through

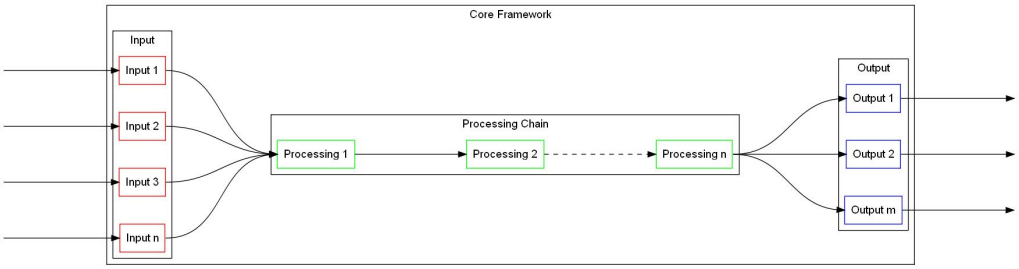

Figure 1: Data flow within the framework. One or more input plug–ins continuously pass data to a chain of processing plug–ins. Resulting data is transferred to multiple output plug–ins.

the cluster node. An application typically consists of multiple, sequential steps. To analyze the data produced by each step (*e.g.* evaluate against ground truth), a method to transparently extract data between processing steps is necessary.

Further Requirements: We want our software suitable for many application scenarios, thus it must also be easy to use for developers and users. Additionally the software needs to be platform independent and thus be applicable in many environments. To provide a powerful but resource–saving platform, we try to minimize the memory and performance overhead caused by the software itself.

3.2. Framework Design

We present a flexible software which fulfills our requirements. Its structure follows the IPO–principle (Input–Processing–Output), i.e. we assume a sequential dataflow through our software.

We have completely implemented a prototype of our design in C#.Net 2.0; an implementation in C++ is currently under construction. The .NET version runs on any platform where a .NET or Mono[1] runtime is available, i.e. at least under Windows, most Linux distributions, Solaris, BSD and MacOS.

3.2.1 Plug–In Structure

To achieve a flexible structure for exchange and combination of single modules we have implemented a plug–in system. Many applications can extend their functionality using plug–ins. Contrary, in our framework the plug–ins actually determine the functionality of the application. Our software defines plug–in interfaces for input, processing and output.

Input plug–ins run in dedicated threads and continuously pass data to the processing chain. The processing part can be composed from multiple processing plug–ins which form a pipeline (further called processing chain). Figure 1 gives an overview on structure and the dataflow within our software.

[1] http://www.mono-project.com/

3.2.2 Generic Data Transfer

For information interchange between plug–ins an asynchronous, event-based mechanism that is independent of data type and content is implemented. Thus, arbitrary data may be processed and passed within the framework as long as the receiving plug–in is able to interpret the incoming data. A computer vision application that works on low level image data will most likely pass images between plug–ins while a fusion algorithm for high–level data may pass only lists of detected objects. We perform a compatibility check to ensure that all plug–receive only supported data.

This enables a wide spectrum of applications and the exchange of single plug–ins without the need to exchange or recompile other parts. In the context of a camera network we can receive the input data for a multi-view fusion algorithm either from synchronized video files or from live data over a network connection (or any other data source). The processing plug–in containing the fusion algorithm does not need to be exchanged when the input source changes. Also for a certain set of input video files we can compare different fusion algorithms by simply exchanging the processing plug–in.

Our software structure and data flow model enables us to use many different communication methods (like low level networking, MPI, CORBA, . . .) in our network.

3.2.3 Monitoring

To achieve a generic mechanism for performance monitoring we provide a monitoring interface which, again, is plug–in based. Every monitor plug–in may listen to events in the system in order to extract performance information. This may be done with or without active support by each plug–in, i.e. plug–ins may define and dispatch certain events that are performance related (e.g. the amount of bytes transferred over a network connection).

Nevertheless, in some cases the method above will fail to collect the necessary performance data. As explained in Section 3.1 we want to analyze the data between each pro-

cessing step. In the monitoring plug–in approach a specific plug–in has to explicitly dispatch a performance–related event containing its processing result. We do not want to unnecessarily complicate the development of plug–ins. Thus, we preferred a much simpler mechanism for this.

We exploit the modular design and the data transfer mechanism to deal with such cases. We insert plug–ins which act as transparent proxies into the processing chain. A proxy plug–in accepts incoming data, performs the necessary operations (dumping or analysis) and passes the unmodified data to the next plug–in. Figure 2 shows an example extraction. If we want to analyze the output of the object detection separately we just insert the corresponding extraction plug–in after the object detection plug–in.

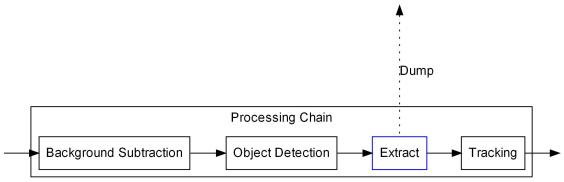

Figure 2: The result of the object detection algorithm is dumped transparently for all other plug–ins.

3.3. Framework Instances

In our framework an application for a certain purpose is composed by selecting plug–ins for input, processing and output. Every plug–in needs to be configured separately, *e.g.* a RTP output plug–in needs a target IP address to stream to, a video file input plug–in requires a source file. A set of fully configured plug–ins that are checked for compatibility and are ready to run is called a framework instance.

Thus, we define a framework instance as a combination of plug–ins

1. that has at least one input and one output plug–in
2. which have been checked for compatibility

$$\{[Vid]^4\} \dashrightarrow \langle [BG] \rightarrow [OD] \rightarrow [OT] \rangle \dashrightarrow \{[Vis], [RTP]\}$$

(a)

$$[VidIn] \dashrightarrow [Transcode_{mpeg4}] \dashrightarrow [VidOut]$$

(b)

Figure 3: (a) A framework instance with four parallel inputs from video files, a processing chain with background subtraction, object detection and object tracking and two parallel outputs to a visualization and to a RTP stream. (b) shows an instance where a simple video transcoding is performed.

3. which are configured and ready to run

To easily describe a framework instance we use the following syntactical elements:

- $[Module]$ describes a single module (plug–in). Multiple parallel modules of the same type are indicated using superscript numbers. Indications on the actual work performed, like the modules configuration, are subscripted if necessary.
- $\rightarrow$ indicates synchronous data transfer between modules (typically inside a processing chain)
- $\dashrightarrow$ indicates asynchronous data transfer from or to a set of parallel plug–ins
- $\{[M1], \ldots [Mn]\}$ describes a set of modules working in parallel (input or output)
- $\langle [M1] \rightarrow \ldots [Mn] \rangle$ describes a processing chain. Due to the strict sequentiality of a processing chain the data transfer typically is synchronous. If a processing chain only contains one element the surrounding brackets may be omitted.

Example framework instances are shown in figure 3. With this description language we can easily give clear descriptions of framework instances.

4. Case Studies

In early experiments we focus on data flows within a single cluster. Embedded devices perform local preprocessing of video data (at 25fps), as CN we use a standard PC platform (2.5GHz, 4GB main memory, Windows). In this case study we perform multi-camera person tracking. Data from cameras is collected at the central node which creates a global probabilistic occupancy map for the common FOV of all cameras[4]. In all setups we perform the same multi-camera algorithm at the CN. Thus, the computational overhead for this algorithm is the same in all setups. Note, that in this work we do not focus on the performance of the algorithm in the CN but rather on the network perspective. The impact of different clustering methods and input data on the performance of multi-camera algorithms will be part of future work.

4.1. Experiment Setups

The embedded devices in our network that are capable of various computer vision tasks. The ”Advanced Video Codec”[2]-devices built by Austrian Institute of Technology are connected to surveillance cameras and perform background subtraction and object tracking on board. Every

[2]http://www.smart-systems.at/products/products_video_avc_en.html

AVC delivers for every camera one RTP-stream containing the raw image, one stream containing only foreground regions and an XML–description of the object tracking output.

Based on the capabilities of our AVCs we evaluate the performance of the cluster node. We have four cameras in our cluster, two connected to each AVC, and perform multi-camera person tracking in the cluster node. In a first scenario the cluster node has raw data as input and performs background subtraction, object detection and multi-view tracking. In second setup the embedded devices stream differential images resulting from the onboard background subtraction to the cluster node. The XML–scene description produced by the AVC is the cluster node input in the third scenario.

Raw images are streamed at CIF resolution, differential images have QCIF resolution, both are in YV12–Format. The XML–description includes metadata like timestamp and input video resolution as well as a list of objects detected in the actual frame and their trajectories. The description for one frame takes approximately one KB of memory depending on the number of objects. Raw images use 148 KB of memory, differential images need 37 KB. Thus, an increased memory demand when using raw data as input must be expected.

We assume approximately 5MB memory overhead for our framework application. This amount differs on every platform, the largest part of this is caused by the graphical user interface. The core framework library itself only uses a few hundred KB. During the execution additional memory is allocated due to the event–based data passing mechanism. Nevertheless, the required memory for this is minimal.

In our experiments we focus on the maximum number of input streams that can be processed on the cluster node. This is important when determining the number of cameras in a cluster and the number of hierarchy levels in a large clustered network.

To switch from one scenario to another we need to change the application in the cluster node. To change the input data or the processing chain itself we only have to exchange plug–ins in our framework. To switch from raw data to differential image we just remove the background subtraction module, to take XML as input we need to change the input plug–in and remove background subtraction and object detection. Changing the plug–in configuration can be done using a comfortable graphical user interface. In this user interface also configuration settings for each plug–in can be changed.

To test our framework we have implemented the following plug–ins:

- RTP-Transport $[RTP]$ (Input)
- XML–Input $[XML]$
- Background subtraction $[BG]$ (uses OpenCV)
- Object detection $[OD]$ (uses OpenCV)
- Multi–view aggregation $[MV]$
- Visualization $[Vis]$

Figure 5 shows the processing chains that were performed in the cluster node depending on the input data coming from the AVC. Obviously, the longest processing chain is needed for raw data. Figure 4 shows the setup of our experiments.

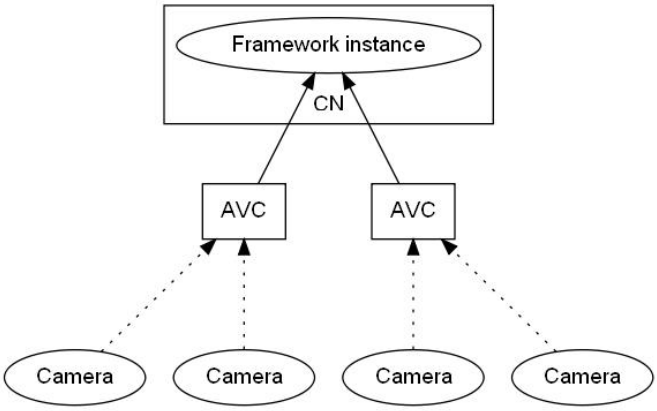

Figure 4: Setup in our experiments. Each AVC is connected to two cameras. The AVC output is streamed to the cluster node and processed in the framework instance.

$$\{[RTP_{CIF}]^2\} \dashrightarrow \langle [BG] \rightarrow [OD] \rightarrow [MV] \rangle \dashrightarrow [Vis]$$

(a) Raw Data

$$\{[RTP_{QCIF}]^4\} \dashrightarrow \langle [OD] \rightarrow [MV] \rangle \dashrightarrow [Vis]$$

(b) Differential Images

$$\{[XML]^4\} \dashrightarrow [MV] \dashrightarrow [Vis]$$

(c) XML

Figure 5: Depending on the input, the cluster node must have an adequate processing chain. The higher abstracted the input, the less work has to be performed by the cluster node.

4.2. Results

As shown in table 1 our cluster node was able to process only two streams of raw data in parallel which is caused by the higher resolution as well as the need to perform background subtraction.

When processing differential images the load on the cluster node decreased drastically. Thus, the cluster node

could process a far higher number of data streams than in our setup.

XML scene descriptions as input for the cluster nodes allow an even higher number of streams to be processed.

Input data	*# Inputs*	*Max. CPU*	*Max. memory*
Raw	2	100%	360 MB
Differential	4	15%	89 MB
XML	4	10%	36 MB

Table 1: Performance indicators wrt. input data (number of input streams processed, Max. CPU load and Max. memory usage).

The bandwidth required to stream a certain kind of data is an important criterion for camera clusters. Table 2 shows the bandwidth values for our setup. Again, the lower load when using higher abstracted data can be seen.

Input data	*Required bandwidth*
Raw	3.7 MB/s
Differential	0.9 MB/s
XML[3]	0.025 MB/s

Table 2: The bandwidth required by every camera to stream the specified data.

As already mentioned, in this case study we did not focus on the algorithms performed in the CN but it can already be seen that in order to employ computationally more demanding algorithms, higher abstracted input or more powerful CN hardware must be employed.

5. Future Work

As shown above, our framework is already a powerful platform for prototyping and evaluating computer vision applications in clusters of cameras.. Nevertheless we will further improve our framework to enable additional usage scenarios.

Currently we are completing the C++ implementation. We can easily port our current implementations from the .NET context to C++ plug-ins since we implemented relevant parts in native C++ already (which are currently wrapped in .NET plug-ins). Therefore it is easy to develop new plug-ins that can be employed in both framework versions.

We are also working on a remote deployment mechanism that will enable us to deploy specific plug-ins and their configurations to other network nodes. This will also require to implement remote control mechanisms for framework instances. The final result will be a network node manager that will deploy and configure network instances on click.

With the network manager we will evaluate networks with multiple clusters to determine the overall performance of a certain clustering mechanism. We will collect different performance parameters in all cluster nodes in the network to determine the optimal clustering for certain scenarios. This includes the number of connected cameras, data abstraction level (inter– and intra–cluster), bandwidth usage and hardware requirements.

We will also focus on modeling multi-cluster networks to facilitate their management and construction.

[3]Note, that the bandwidth required to stream XML scene descriptions varies with the number of objects in the steam. One object in one frame requires approx. 105 bytes.

References

[1] H. Aghajan and A. Cavallaro. *Multi-Camera Networks - Principles and Applications*. Elsevier, 2009. 1

[2] H. Detmold, A. van den Hengel, A. Dick, A. Cichowski, R. Hill, E. Kocadag, K. Falkner, and D. S. Munro. Topology Estimation for Thousand-Camera Surveillance Networks. In *Proceedings of the ACM/IEEE Conference on Distributed Smart Cameras*, Vienna, Austria, 2007. 1, 2

[3] S. Fleck, F. Busch, P. Biber, and W. Straßer. 3D Surveillance A Distributed Network of Smart Cameras for Real-Time Tracking and its Visualization in 3D. In *CVPRW '06: Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*, page 118, 2006. 1, 2

[4] F. Fleuret, J. Berclaz, R. Lengagne, and P. Fua. Multicamera People Tracking with a Probabilistic Occupancy Map. *IEEE Transactions and Pattern Analysis and Machine Intelligence*, 30(2):267–282, 2008. 4

[5] S. Funiak, C. Guestrin, M. Paskin, and R. Sukthankar. Distributed Localization of Networked Cameras. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 34–42, 2006. 1

[6] H. Medeiros, J. Park, , and A. C. Kak. A light-weight event-driven Protocol for Sensor Clustering in Wireless Camera Networks. In *Proceedings of the ACM/IEEE Conference on Distributed Smart Cameras*, pages 203–210, Vienna, Austria, 2007. 1

[7] B. Rinner and W. Wolf. Introduction to Distributed Smart Cameras. *Proceedings of the IEEE*, 96(10):1565–1575, October 2008. 1

[8] A. Williams, D. Ganesan, and A. Hanson. Aging in Place: Fall Detection and Localization in a Distributed Smart Camera Network. In *MULTIMEDIA '07: Proceedings of the 15th International Conference on Multimedia*, pages 892–901, 2007. 1