Facet: Towards a Smart Camera Network of Mobile Phones

Philipp Bolliger
bolligph@inf.ethz.ch

Moritz Köhler
koehler@inf.ethz.ch

Kay Römer
roemer@inf.ethz.ch

Institute for Pervasive Computing, ETH Zurich, Switzerland

ABSTRACT

Smart camera networks provide the opportunity to detect, classify, and trace visual events by means of a wireless network of embedded computing devices equipped with camera sensors. Previous research in this area has largely focused on custom hardware solutions. In contrast, we propose to use mobile phones as a portable and low-cost platform to implement smart camera networks. Due to mass production, the price of mobile phones is constantly dropping while additional functionality is being added. In particular, the addition of cameras and ad hoc networking enable the use of mobile phones for smart camera networks. Also, software developed for mobile phones is highly portable due to standardized programming environments and APIs.

In this paper, we present *Facet*, a self-organizing network of smart cameras and its software architecture, which addresses the specific challenges of the mobile phone platform. An initial prototype of Facet has been developed and is used to obtain first performance results in an office setting.

Keywords

Smart Camera Networks, Wireless Sensor Networks, Mobile Phones.

1. INTRODUCTION

Recently, smart distributed cameras have become an active field of research. Here, small embedded computing devices equipped with CMOS cameras form a wireless network to detect, classify, and track visual events in the environment. Smart distributed cameras are envisioned to be applied for surveillance, traffic management, health care, assistance of elderly people, environmental monitoring and others [1].

The research focus in this area has been on high-performance on-board computing and communication infrastructure, as well as a deliberate combination of computer vision, video sensing, and communication tasks (e.g., [13, 24]). Also software architectures and frameworks have been proposed (e.g., [4]) of which most are more or less bound to the underlying hardware architecture.

In contrast to these research directions, we propose to use existing mobile phones as a foundation for smart distributed cameras. In particular, we develop a portable software framework called *Facet* to facilitate the development of applications based on smart camera networks of mobile phones.

This approach is partially motivated by a forecast that sees a slightly different interpretation of Moore's law for the mobile-phone industry [17]: Instead of simply increasing processing power or memory capacities, processors for mobile phones undergo a constant increase in functionality, like the most recent additions of mp3 and video streaming. In the future, mobile phones might have specialized functionalities for object recognition and video processing to enable easier real-world interaction. This development and the fact that worldwide mobile phone sales almost have reached 1bn in 2006, of which 48% were camera phones, and 70-80% Java enabled [11], brings us to a very simple, but somewhat provoking idea: Instead of building a very specific platform for a network of smart cameras, we could use the mobile phone as a cheap, widely accepted, easy to program, and ubiquitous platform for developing networks of smart cameras. If we can use mobile phones to build smart camera systems, we will be able to build and deploy a system that enables scenarios such as object recognition, object tracking and the like at a fraction of the costs of traditional special purpose systems. In addition, maintenance, and upgrading of the network as a whole becomes easier since we can build on well established standards for communication such as Bluetooth, UMTS, or WIFI. Moreover, the deployment of such smart camera networks would not only be cheap but also very easy once the required software components are commonly available. It could also open the road for new business cases for mobile phone producers which then could offer very slim and simple devices not even meant for communication between people, but for all sorts of applications.

In the remainder of the paper, we first sketch some motivating application scenarios in Sect. 1.1 and discuss opportunities and challenges of using mobile phones for smart camera networks in Sect. 1.2. In Sect. 2 we outline the architecture of our software framework *Facet*, before presenting a first prototype implementation of *Facet* in Sect. 3. We evaluate important aspects of this implementation in Sect. 4.

1.1 Application Scenarios

Our focus is mostly on indoor applications, where mobile phones can be easily deployed by placing them on furniture or attaching them to the ceiling or walls. For applications that require a lifetime of more than a few days, phones need to be connected to a permanent power supply.

In such a setup, smart phone networks can be used to detect and report certain visual events (e.g., people entering rooms during the night), for statistical applications (e.g., occupancy of certain places over time), and for tracking applications (e.g., to locate certain objects or to guide people to certain places).

Realizing these applications requires a number of basic functionalities. First of all, individual nodes must *capture and analyze* a stream of images from the camera to detect certain visual events. In many cases, many phones must *cooperate* and exchange information in order to realize an application. Such cooperation often requires that visual events observed by different phones have to be examined in a common temporal and spatial reference system to answer questions such as *"Was event X observed before event Y?"* or *"Were events X and Y observed in the same room?"*. Answering these questions requires services for *time synchronization* and *location calibration*.

Our framework *Facet* provides portable implementations of these services as discussed in Sects. 2 and 3, taking into account the specific properties of mobile phones as discussed in Sect. 1.2.

1.2 Opportunities and Challenges of using Mobile Phones

Using mobile phones to implement a smart camera network presents both opportunities and challenges. In contrast to existing smart cameras, mobile phones provide a relatively standardized programming interface based on the J2ME implementation of Java, including standardized APIs for Bluetooth (JSR-82 [28]) and ad-hoc networking (JSR-259 [27], still under development), or web services (JSR-172 [8]). This enables the provision of a highly portable software infrastructure that can be be executed by a wide variety of mobile phones from different vendors. On the other hand, these standardized APIs sometimes do not provide access to certain functions of a mobile phone. For example, the Bluetooth API does not provide access to the HCI layer of the Bluetooth stack, which complicates the implementation of functions such as time synchronization (see Sect. 3.3 for details).

Many existing smart camera implementations provide plentiful computational, storage, and communication resources. Some platforms even provide specialized digital signal processors [5] to run computer vision algorithms. Although mobile phones are becoming more capable over time as well, the resources provided by currently available devices are small compared to that of many specialized smart cameras. Also, while Java provides a portable execution environment, performance often suffers from bytecode interpretation in the Java VM.

Similar restrictions apply to the capabilities of the camera modules built into mobile phones. Often, APIs offer limited or no control over the optical parameters (e.g., focus, zoom, blind) and the orientation of the camera.

Typical mobile phones offer multiple technologies for wireless networking, for example Bluetooth for forming ad-hoc networks among nearby phones and other Bluetooth devices, as well infrastructure-based long-range communication such as GPRS. These capabilities can be exploited to form flexible smart camera networks. In particular, Bluetooth can be used to form local ad-hoc networks of nearby phones with high bandwidth and free of charge, while infrastructure-based communication can be used to interconnect distributed ad-hoc networks. Infrastructure-based communication can also serve as a gateway between smart phone networks to the Internet.

2. ARCHITECTURE

To realize the applications sketched in Sect. 1.1, a number of basic services are required that include computer vision aspects, wireless communication, as well as support for time and location management. Our goal is to provide a software framework that offers these basic services as a foundation for the development of applications based on smart camera networks of mobile phones. In this section we outline the architecture of our software framework called *Facet*.

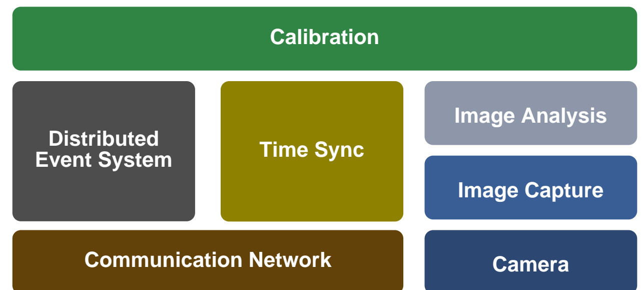


Figure 1: Overview of *Facet*'s architecture.

Fig. 1 shows an overview of *Facet*'s architecture. Each mobile phone captures a stream of images from the camera and analyzes these images to detect different vision events, such as an object entering or leaving the field of view of the camera. These events have a type (e.g., *in* or *out*, i.e., an object entering or leaving the field of view) and parameters such as the identity of an object, the point in time when the event has been detected, or the location where the event has been detected. The limited computational resources of mobile phones require that the algorithms used for detection of vision events are rather simple and efficient.

Many applications require the collaboration of multiple mobile phones. As the bandwidth of wireless communication among mobile phones is rather constrained, collaboration in our framework is based on distributed events (rather than on the exchange of raw image streams). We adopt a publish/subscribe approach here, where a node can issue a subscription to certain types of events from phones in a certain network neighborhood. An example subscription could for example declare interest in all *in* events generated by phones that have a direct network link to the subscriber. As soon

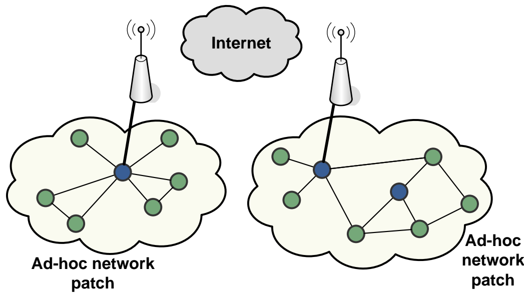


Figure 2: Interconnecting ad-hoc network patches.

as a node generates an event, this event is delivered to all matching subscribers.

A specific example of collaboration of multiple phones is to examine correlations among vision events generated by different phones. Here, it is often necessary to know the time when an event has been observed and the location where this event has been observed. For example, to estimate the velocity of a tracked object, we can correlate an *out* event $eout(o, t_1, l_1)$ and an *in* event $ein(o, t_2, l_2)$ that refer to the same object o which left the view of a camera at time t_1 and location l_1 and entered the view of another camera at time t_2 and location l_2. When can then estimate the velocity of the object by computing $|t_2 - t_1| / |l_2 - l_1|$. This of course requires that timestamps t_1 and t_2 refer to a common time scale, which requires time synchronization. Likewise, locations l_1 and l_2 have to refer to a common coordinate system, which requires camera calibration.

In the following subsections we discuss certain aspects of the architecture in more detail.

2.1 Communication Network

We consider a heterogeneous network architecture as depicted in Fig. 2, where nearby phones form an ad-hoc network *patch* using infrastructure-free short range communication such as Bluetooth. These patches are interconnected among each other by means of infrastructure-based long range communication such as GPRS. The latter may also be used to connect one or more patches to other networks such as the Internet. For example, a server connected to the Internet could subscribe to vision events from one or more network patches and analyze these to track objects as they move from phone to phone and from patch to patch.

The idea behind this network architecture is that phones in the same patch typically require strong collaboration (which is supported free of charge by high-bandwidth ad-hoc networks such as Bluetooth), whereas there is a more loose and infrequent collaboration between phones that are part of different patches (which uses chargeable services with lower bandwidth such as GPRS).

2.2 Time Synchronization

Phones within a patch require time synchronization to establish a common time scale. For the purpose of our framework, *internal* synchronization suffices. That is, nodes need to agree on *any* common time scale rather than on a specific one such as UTC. Also, as we are focusing on indoor applications, most vision events will be triggered by human mobility. As the velocity of humans is in the order of few meters per second and the spacing between phones is in the order of meters, a synchronization accuracy in the order of tens of milliseconds would be sufficient to reliably establish a temporal ordering of vision events triggered by a single human walking across a space instrumented with smart cameras.

One specific difficulty with time synchronization using J2ME on mobile phones is the lack of an API to adjust the system clock to a common time scale. Hence, time synchronization in *Facet* is based on *timestamp transformation* [22]. Here, each mobile phone timestamps events using its unsynchronized local clock. Before sending this timestamp to another phone, the timestamp of the event is transformed to the time scale of the receiving node.

2.3 Camera Calibration

Camera calibration is concerned with estimating the pose (i.e., location and orientation, which results in 6 degrees of freedom) of each camera in a common coordinate system. While in principle it would be possible to perform calibration by placing each camera in a predefined way, this is often impractical even for small networks. Hence, we need an automated way to calibrate a patch of cameras. In general, all approaches can be separated into two groups: calibration based on visual evidence, such as overlapping fields of view or markers, and calibration based on motion of objects like people passing by an observed area. The latter approach is suitable to calibrate non-overlapping cameras. As we want to support setups with non-overlapping cameras, we employ the latter technique in *Facet*.

Calibration based on moving objects is an active field of research. Several algorithms have been proposed (e.g., [19], [10]), but several challenges remain. Hence, rather than relying on a specific algorithm, we provide an open interface that allows to use different algorithms. The foundation for this open interface is the observation that existing algorithms are based on spatial and temporal constraints between cameras (e.g., camera X and Y are at most Z meters apart). These constraints are derived from vision events triggered by mobile objects (e.g., if the maximum speed of a mobile object is known, then the time interval between successive detections of this object by different cameras can be used to derive an upper bound on the distance between these cameras).

To support calibration algorithms based on constraints, *Facet* provides a constraint graph. Cameras in a network patch form the nodes of the constraint graph, while constraints as in the above example form the edges of the graph. Each constraint has a type (e.g., maximum-distance constraint) and zero or more parameters (e.g., the maximum distance in meters). *Facet*'s constraint graph is an extension of spatial relationship graphs [18] with support for additional constraint types derived from the network topology (e.g., two nodes are connected by a *link* constraint if the cameras have a direct communication link). The constraint graph is implemented in a distributed fashion in *Facet*: each graph node (i.e., mobile phone) stores its adjacent edges.

In summary, calibration with *Facet* requires mobile objects

to move through the area being observed by the camera network. These objects trigger vision events that result in edges in the constraint graph. When enough constraints have been collected, a calibration algorithm is executed on the constraint graph to compute the pose of each camera.

3. IMPLEMENTATION

To study the feasibility of our approach, we produced an initial implementation of *Facet* with basic functionality. Although conceptually straight-forward, many functions required unconventional approaches due to limitations of the phone platform. In this section, we describe our solutions.

3.1 Communication Network

Our prototype implementation currently supports a single ad-hoc patch. Nodes within the patch form a multi-hop network using Bluetooth. One node in the patch is designated as a master node, which maintains a GPRS connection to deliver data to a PC. Within the patch, unicast routing of messages between any pair of nodes is supported.

Although several Java networking frameworks exist, most of them are not usable in the context of our work. For example, JXME [29] does not support Bluetooth, and JORAM [26] requires a central messaging platform. Hence, we worked with the basic Bluetooth API JSR-82 [28]. Unfortunately, JSR-82 has the fundamental limitation that only a single Bluetooth connection can be open at a time. To send a Bluetooth message to a new destination node, any open connection has to be closed first before being able to open a connection to the new destination.

Another problem is that Bluetooth's inquiry procedure to find out the addresses of network neighbors takes quite a long time. In our experiments, an inquiry took at least 15 seconds to complete, but may last up to 2 minutes. Hence, the use of this function should be reduced to the absolute minimum.

For our solution, we assume that the patch network is static, that is, the network topology is fixed. To support message routing in such a patch network, we use a modified version of the algorithm presented in [3] as depicted in Alg. 1. Each node maintains two variables, the set of network neighbors *ntable* and a routing table *rtable* which contains an entry for each node d in the patch network, such that $rtable[d]$ holds the address of the next hop on the path to d. Upon initialization, a node performs a Bluetooth inquiry to discover the addresses of its neighbors, which it stores in *ntable*. Then it sends an *announce* message to each of its neighbors to announce its presence. A node which receives this message includes the sender in its neighbor table and creates an appropriate entry in its routing table for this node. Then the node forwards the *announce* message to its neighbors, allowing them to create a routing table entry for the new node as well. This process terminates when all nodes in the patch have created an entry in their routing tables for the new node. Finally, the immediate neighbors of the new node reply a *route* message containing their routing table entries to allow the new node to create appropriate entries in its routing table. Note that this approach does not necessarily result in "best" (e.g., shortest) routing paths, but the algorithm can be easily extended to obtain shortest paths using a distance vector approach.

Once all nodes have joined the network and nodes have setup their routing tables using the above algorithm, nodes can send messages to any other node in the patch. For this, the source node first checks if the destination node is a direct neighbor and sends the message directly in this case. Otherwise, the source node consults its routing table to find the next hop and forwards the message to this node. This process is repeated until the message reaches its final destination.

Sending a message to a neighbor node n is implemented as follows. If there is an open connection to n already, then the message is sent over this connection. Otherwise, if there is an open connection to a neighbor other than n, then the connection is closed, a new connection to n is opened, and the message is sent. Otherwise, if there is no open connection, then a connection to n is opened and the message is sent. Opening and closing a connection to a node with known Bluetooth address took less than 1 second in our experiments.

```
on init:
ntable = inquiry();
foreach n ∈ ntable do
    send announce<self> to n;
    receive route<rt> from n;
    foreach r ∈ rt do
        rtable[r] = n;
    end
end

on receive announce<dst> from s:
ntable = ntable ∪ s;
if dst ∉ rtable then
    rtable[dst] = s;
    foreach n ∈ ntable \ s do
        send announce<dst> to n;
    end
end
if dst = s then
    send route<rtable> to s;
end
```

Algorithm 1: Routing algorithm for a Bluetooth patch.

3.2 Distributed Events

In our initial prototype, we implemented a very simple form of event distribution. When a node generates an event, this event is broadcast to all nodes in the patch using the routing algorithm given in the previous section. In addition, the master node sends the event to the backend PC via GPRS.

3.3 Time Synchronization

As Bluetooth uses a time division multiple access scheme for communication, it maintains an internal clock that is tightly synchronized between nodes that share a connection. We exploited this feature in [21] to measure the offsets Δ_{ij} between the system clocks of neighboring nodes i and j. Time synchronization is then implemented by transposing timestamps as follows. Node i first obtains a timestamp t_i using its local unsynchronized clock. When t_i is sent to node j, the timestamp is transposed by adding the clock offset

Δ_{ij} to t_i. This approach works also across multiple hops and has two primary advantages: it avoids the difficulties of explicitly synchronizing the system clocks and does not require any master time.

Unfortunately, the Bluetooth API of the connected limited device configuration (CLDC) [25] on mobile phones does not provide access to the Bluetooth clock. Hence, we need a different way to measure the clock offsets Δ_{ij} between network neighbors. Since the requirements on synchronization accuracy in *Facet* are not very strong as pointed out in Sect. 2.2, we resorted to the following simple scheme, which is executed for each pair i, j of neighboring nodes. Node i reads its local clock, obtaining a timestamp t_i and sends this timestamp to node j using a *sync* message. At arrival, node j reads out its local clock obtaining a timestamp t_j. We then compute $\Delta_{ij} = t_j - t_i$, neglecting the (nonzero) message latency. However, as the latency is very variable, the clock offset obtained this way differs heavily from measurement to measurement. We therefore perform k such message exchanges, obtaining k different clock offset values. In our experiments, we used $k = 20$. Among these values, we select the smallest one, since for the respective measurement the message latency was the smallest. Finally, the receiving node sends the computed clock offset to the sending node.

In our prototype implementation, we perform this synchronization procedure once for every pair of neighbor nodes at startup. However, due to clock drift, the accuracy of synchronization will decrease over time. To support systems with a longer lifetime than in our initial experiments, synchronization would have to be repeated from time to time.

3.4 Image Capture

Capturing an image from the camera seems to be a straightforward task at first. However, to our surprise we had to find out that capturing an image on a mobile phone, and only having the J2ME API available, is harder than it seems. Again, the main problem is speed. To reduce the effort required to process an image, we found that we could reduce the image size down to 120 by 160 pixels without loosing accuracy in the image analysis process. But still, the process of capturing an image took more than 1 second on the mobile phones used (see Section 4 for details) because the standard capturing routine creates a jpeg or png image. However, the API also allows to capture raw images stored in a byte array. This approach showed to be about twenty times faster. In this byte array, each pixel is encoded in three bytes: one for red, green, and blue, respectively. As we don't need the color information but only gray values, we convert the values in one single loop thus reducing the time needed to capture an image to about 80 milliseconds.

3.5 Image Analysis

Our prototype implementation supports the detection of two types of events: *in* events of objects entering the field of view of a camera and *out* events of objects leaving the field of view again. In particular, we consider humans as "objects" in our experiments. These events will be used to infer constraints for calibration as described in Sect. 3.6. Due to the limited performance of our communication architecture, these events should be detected on the phone itself, such that we avoid communication of raw images between camera nodes.

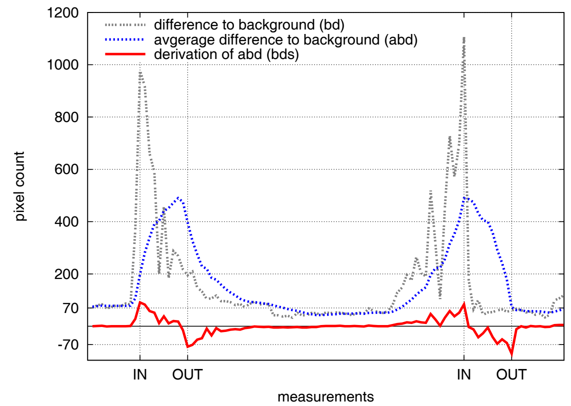


Figure 3: Exemplary background subtraction.

The basic approach to detect these events is to compute the difference between the current image and a previously stored background image that contains no moving objects. To calculate the difference between the current image and the stored background image, we subtract the images pixel by pixel. If the difference is greater than a certain threshold value, whose value we determined experimentally, we increment a background difference counter *bd*. To speed up image analysis, we combined gray value conversion (see Sect. 3.4) and background subtraction into a single loop.

Figure 3 shows a typical graph of the background difference counter *bd(t)* over time t. The first peak indicates a person passing through the field of view in one direction whereas the second peak indicates a person returning, i.e., passing through the field of view from the other direction. The graph shows a notable oscillation at the falling edge of the first peak and at the rising edge of the second peak. This is due to the camera of the mobile phone automatically correcting the white balance slower than our image analysis is capturing and processing images. To remove these high frequency components, we apply a simple moving average filter to the last 10 values, i.e., $abd(t) = 1/10 \sum_{i=0}^{9} bd(t-i)$. Although this approach delays the event generation by about 1 second, it helps making the image analysis much more robust.

The absolute values of the peaks for people moving through the field of view heavily depend on the light conditions, clothing, as well as speed and trajectory of movement. However, we observed that the maximum slope of $abd(t)$ is remarkably independent of these conditions. Hence, we compute an approximation of the first derivation of $abd(t)$ as follows: $bds(t) = abd(t) - abd(t-1)$. See Fig. 3 for an example. To detect *in* and *out* events, we apply thresholding to $bds(t)$. We empirically found a threshold of $\pm$ 70 to work well. That is, if $bds(t)$ rises above 70, then an *in* event is detected. If $bds(t)$ falls below -70, then an *out* event is detected.

In order to accommodate to changes of the background image, e.g., due to changing light conditions, we also implemented a very simple background image adaption algorithm. To detect durable changes of the background we applied the following rule: if the difference from the background is sub-

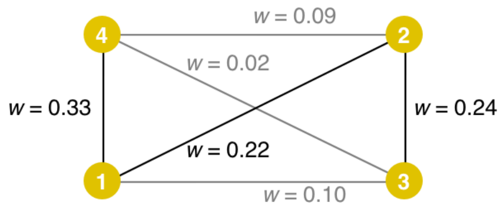


Figure 4: Exemplary constraint graph before edge removal.

stantial (i.e., $bd(t) > T_{bd}$) and the difference from image to image is small (i.e., $bds(t) < T_{bds}$) for at least T_N successive t, then we update the background image. We empirically found threshold values $T_{bd} = 90, T_{bds} = 10, T_N = 100$ to work well.

3.6 Calibration

In our prototypical implementation, *in* and *out* events are used to derive constraints to populate the constraint graph. Here, we derive visual neighborhood constraints between pairs of nodes. Two nodes a and b are said to be visual neighbors if tracked humans walk from the field of view of a to the field of view of b without passing through the field of view of any other node. In the setup depicted in Fig. 5, for example, where four cameras are linearly aligned in a hallway, nodes 1 and 2 are visual neighbors, but not nodes 1 and 3. Visual neighbors are connected by an edge in the constraint graph.

To detect visual neighborhood, we identify *matching pairs* of *out* and *in* events. A pair (o, i) is said to be a matching pair if o is an *out* event, i is an *in* event, and no other event occurred after o and before i. For each such matching pair, we create an undirected edge (assuming that visual neighborhood is a symmetric relation) in the constraint graph between the node that generated event o and the node that generated event i. Each edge between nodes a and b is annotated with a counter c_{ab} that counts the number of matching event pairs for this edge. The weight w_{ab} of an edge between a and b is defined as $w_{ab} = c_{ab} / \sum_{e,f} c_{ef}$.

As explained in Sect. 2.3, the constraint graph is distributed in the sense that each node stores its adjacent edges. To implement this, every event is broadcast to every node in the patch as described in Sect. 3.2. Every node then uses the approach described in the previous paragraph to find matching pairs of events which involve a locally generated event, as these represent edges that are adjacent to the node.

Consider the constraint graph in Figure 7 as an example. If node 2 broadcasts an *out* event, and shortly after that node 1 broadcasts an *in* event, then both nodes update their local graph by incrementing c_{12} by one. Figure 7(a) shows the constraint graph after two matchings have been detected, namely the one described above and a second one between node 1 and node 4. Thus, the edge weight w is 0.5 for both edges.

False negative and false positive events cause our approach to wrongly create edges between nodes that are actually not visual neighbors. For example, the constraint graph in Fig. 4 contains edges between nodes that cannot be visual neighbors according to Fig. 5. In fact, only the edges between the nodes 4 and 1, between 1 and 2, and between 2 and 3 are *true* edges. All the other edges are said to be *false*.

To remove false edges, we assume that false negative/positive events are generated rarely and thus the weights of false edges will be small compared to those of true edges. In our prototype implementation, we assume that all camera nodes detect about the same number of events. We will see in Sect. 4.5 that these assumptions are reasonable for our experiments. The edge removal procedure first counts the number n of edges in the constraint graph. With the above assumptions, true edges should have a weight w that is substantially larger than $1/n$, while false edges can be expected to have a weight substantially smaller than $1/n$. That is, we remove all edges with a weight smaller than $1/n$. In the example graph in Figure 4, there are $n = 6$ edges, so all edges with a weight smaller than $1/6$ will be removed, resulting in the graph depicted in Fig. 7(b).

4. EVALUATION

To verify the feasibility of a camera network of mobile phones, we performed a preliminary evaluation of the crucial components of the *Facet* prototype described in Sect. 3. In particular, we investigated the latency of Bluetooth-based communication, as well as the the accuracy of time synchronization, image analysis, and calibration. All tests were conducted using *Nokia 6630* mobile phones having firmware 6.03.40 installed. As the 6630 is a rather old mobile phone – it was released in November 2004 – better results can be expected for more up-to-date hardware.

4.1 Experimental Setup

For the evaluation we performed experiments with four mobile phones in our office hallway. The nodes were attached to the ceiling and used to detect people passing by. In a first setup, we aligned the mobile phones *linearly*, mounted in equal distances of 3.7 meters as illustrated in Figure 5. The second setup, intended to be more realistic and thus more demanding, consisted of three linearly aligned nodes in the hallway and one node in the printer and coffee machine room in a *triangular* setup as depicted in Figure 6. Moreover, we increased the distance between node number 4 and node number 1 to 7.4 meters.

To analyze the formation of the constraint graph, the nodes where programmed to send all events via GPRS to a central server in addition to exchanging them among each other via Bluetooth. Both setups, i.e., the linear and the triangle experiment, where used to collect events during normal business hours, i.e., not during lunch or over night. We gathered more than 8 hours of data for the linear setup, and another two hours for the triangular setup.

4.2 Communication

As described in Section 2.1, we use Bluetooth to organize nodes in local ad-hoc patch networks and GPRS to interconnect multiple patches. In the following, we will discuss the performance of Bluetooth communication and refer to [14] and [16] for evaluation results on GPRS channel allocation.

Figure 5: Linear camera setup in a hallway.

Figure 6: Cameras setup in triangle.

The message latency for Bluetooth communication in our prototype is mainly dominated by opening and closing a Bluetooth connection. Our experiments with Nokia 6630 phones showed that when connecting directly to other phones with a known Bluetooth address, small messages ($<$ 20 KB) can be sent in under 2 seconds, the time to establish and close the link connection included.

In our experiment setups, the diameter of the network patch is at most 3 hops, hence it takes at most 6 seconds to send a message through the network. Although this result is very promising compared to [3], it also imposes a limit on the speed of objects that can be tracked in realtime. With an average distance of 4 meters between neighboring nodes in the linear setup, the per-hop message latency will be larger than the time it takes an object to move from one node to another if the object moves faster than 2 m/s. However, in our experiments humans went very rarely faster than 1.7 m/s in a hallway.

We omit a detailed performance analysis of the algorithm proposed to disseminate the Bluetooth device addresses as it only needs to be run once when initially setting up the camera network and thus the overhead is negligible. Yet, we can say that in our four node experiment setup (see Figure 5), the distributed algorithm rarely needed more than 25 seconds to complete. This approximately equals the time needed to complete a Bluetooth inquiry (15-20 seconds) and to send 3 consecutive *announce* messages (2 seconds each).

4.3 Time Synchronization

To measure the clock offsets between neighboring nodes, each pair of neighbors exchanges a sequence of messages without closing the connection in between. As connection setup takes far longer than sending a message, the time required for synchronization of a pair of nodes is about the same as sending a single message, i.e., less than 2 seconds. But, as we cannot synchronize nodes in parallel due to the limitations of the used Bluetooth API, this number has to be multiplied by the number of neighboring node pairs in the patch. In our hallway test using four nodes, the time synchronization took between 6 and 8 seconds.

The question remains how accurate the clocks can be synchronized. By executing the synchronization algorithm multiple times in sequence and measuring the difference between the resulting clock offsets, we found the values to vary by about 30 milliseconds. In addition, there is a systematic error due to the fact that we do not take into account the message latency. We could not quantify this error due to the lack of an external reference system for comparing the clocks of two phones.

4.4 Image Analysis

The task of the image analysis component, which runs locally on each node, is to generate the appropriate *in* and *out* events. Hence, camera images must be processed fast enough to track humans passing by. Likewise, the image analysis component should only generate correct events in order for the calibration algorithm to generate the correct constraint graph. Thus, the image analysis component should not create false positive events, i.e., send an event although no human passed by, as this would distort the calibration by leading to false matchings (see Sect. 3.6). Even worse are false negatives, i.e., not sending an event although a human was passing by, as they not only distort the calibration but also delay it.

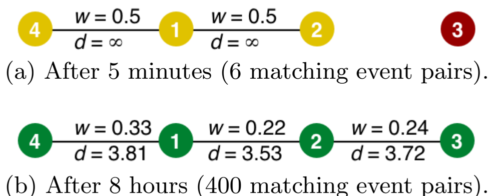
To test the accuracy with which the image analysis component generates *in* and *out* events, we gathered all events centrally on a server and compared them to the ground truth, that is a record of several members of our group walking through the hallway and going to the printer room, respectively. Thereby, we found that no mobile phone generated false positives while nearly 20% false negatives occurred. As expected, false negatives mainly occurred when two or more people entered the field of vision of a camera at about the same time and subsequently left one after another. Very infrequently, false negative events occurred because people ran through the hallway, thus generating a background difference derivation (*bds*) smaller than the threshold. Due to the limited image size, low resolution and in particular due to the low computing power available on the mobile phones used, we were not able to improve the image analysis so as to count the number of people concurrently in the field of vision. For the same reasons, we could not further speed-up the image processing on the Nokia 6630 phones and were thus only able to track objects or humans not faster than 2 m/s. On the upside, we found that although different clothing colors and differences in their brightness did result in quite different *bds* peaks, we were still able to generate the correct events by adapting the threshold accordingly.

As mentioned in Section 3.5 and found in the above section, one of the biggest challenges developing the image analysis component was to speed-up the analysis per se. Our first naive implementation of background subtraction was capable of processing about 0.75 frames per second. Taking into account that a human passing by will remain in the field of vision for only about 2.5 seconds, it became clear that this was way too slow. However, by applying the different optimizations detailed in Section 3.5, we finally achieved a frame rate of 12.8 fps (average frame rate measured during

both the linear and the triangle experiment).

4.5 Calibration

Our framework supports the construction of a constraint graph as described in Sect. 3.6, which is intended to be used as the input to an calibration algorithm which computes the locations of cameras. Hence, the accuracy of the constraint graph is crucial for the accuracy of the final calibration. Here, the accuracy of the constraint graph is informally defined as the level of congruence between the actual layout of the camera nodes and the constraint graph. For both experiments, our approach produced correct constraint graphs despite varying environmental parameters such as light intensity, clothing, or velocity. The resulting graphs (after removal of edges with a weight below the topology-specific threshold) are shown in Figs. 7(b) and 8, respectively.



(a) After 5 minutes (6 matching event pairs).

(b) After 8 hours (400 matching event pairs).

Figure 7: Constraint graphs for the linear setup.

Besides the topology of the constraint graph, we also tried to infer the length of the edges (i.e., distance of camera nodes that are visual neighbors). For this, we consider the time interval between a matching pair of *out* and *in* events and multiply this value with an average human walking speed. We used a value of 5.4 km/h (i.e., 1.5 m/s) according to [6]. In Figs. 7(b) and 8, graph edges are annotated with the resulting distance estimates. For edges where people walk at constant speed (e.g., along the hallway), the resulting distance estimates are quite accurate. For these edges, we obtain an accuracy of ± 17 centimeters in the linear setup and ± 44 centimeters in the triangle setup. The larger error for the triangle setup is due to the increased distance between nodes 1 and 4. Overall, this results in a relative error of about 5-6%. However, for the edge between nodes 1 and 3 in Fig. 8, where people change walking speed to enter/leave the printing room, the distance estimate has very poor accuracy.

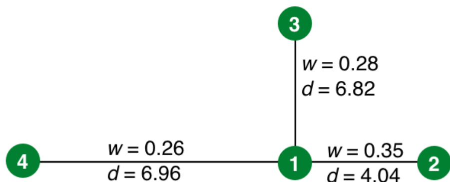


Figure 8: Final constraint graph for the triangle setup.

As the constraint graph changes over time with every new matching pair of *out* and *in* events, we are interested in how the quality of the constraint graph evolves over time. For this purpose, we consider the *confidence* of the constraint graph, which is defined as the sum of the weights of the true

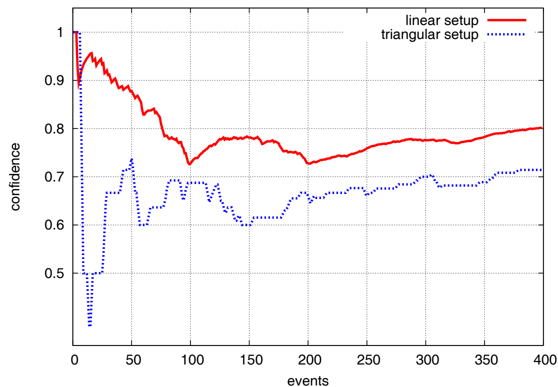


Figure 9: Development of graph confidence over time.

edges. Fig. 7(a), for example, shows the constraint graph for the linear setup after 6 matching event pairs have been processed, which all refer to true edges. Here, the confidence is 1.0 or 100%.

However, failure of the image analysis component to detect certain *in* or *out* events will result in false edges being added to the constraint graph, such that the confidence decreases. As more and more events are being processed, we expect the confidence value to converge to a stable value due to the law of large numbers. Figure 9 illustrates the development of the confidence over time for both the linear and triangular setups.

Both curves show a similar qualitative behavior, starting at a value of 1 since the probability for a matching pair of events that refers to a true edge is larger than the probability for a matching pair of events that refers to a false edge. Over time, wrong edges appear, so the confidence values drop. Slowly, the confidences converge to a stable value as expected. In general, the probability of false matchings is larger in the triangle setup than in the linear setup, resulting in lower confidence values for the former experiment. In the linear experiment, confidence first drops to 0.725 before converging to about 0.8 after 400 matchings, which perfectly coincides with the false event detection rate of about 20% as reported in the previous section. In the triangle experiment, confidence drops to as low as 0.4 before converging to about 0.7.

5. RELATED WORK

Recently, several embedded hardware platforms equipped with vision sensors have been developed (e.g., [7, 13, 20]) and supporting software frameworks have been devised. While the architecture of these software platforms bare some similarities to *Facet*, these are custom solutions for specific platforms. In contrast, *Facet* is aimed to support mobile phones as a standardized and portable hardware platform.

Camera calibration is a very well explored and understood topic in computer vision. For example, [12] give an excellent introduction into the field.

Baker et al. [2] focus on classic calibration of multi-vision

systems using textures such as checkerboards. Their approach results in high precision calibration, however, it requires a great amount of user attention. Additionally, every time a node is replaced, calibration has to be redone at least locally, which results in high maintenance cost.

Jannotti et al. [15] propose a calibration based on Geographic Hash Tables, however, their work is still limited to 3D reconstruction from overlapping fields of view which cannot always be guaranteed.

Taylor et al. [23] propose a calibration and positioning technique based on light sources attached to cameras which then can be viewed and analyzed by each camera, respectively. This approach again can lead to high precision. However, since cameras need to be able to "see" each other, this approach requires overlapping fields of view.

In [9], Ercan et al. investigate object tracking via camera networks, especially in situations where occlusions occur. Their approach is similar to ours, since they put an emphasis on simple image preprocessing on the camera node, in order to reduce network traffic. However, they assume an already calibrated system and do not address reconfigurability of the camera network.

Different from vision calibration but of similar importance is the calibration of non-overlapping cameras. The goal is to achieve spatio-temporal relationships between cameras. Rahimi et al. [19] establish these links by modeling the trajectory of objects with Gaussian Markov chains. Their approach is the most similar one to ours. An empiric comparison of the two approaches therefore would be interesting.

6. CONCLUSION

In this paper we have presented *Facet*, a software architecture to support smart camera networks of mobile phones. In contrast to other work which relies on custom-built hardware, our approach is based on low-cost mobile phones with standardized Java APIs. We have implemented a prototype of *Facet* and showed through a preliminary evaluation the feasibility of this approach. In particular, we support multi-hop Bluetooth networking, time synchronization, detection of visual events on the phone itself, as well as first steps towards calibration. Although our system is executed on a Java Virtual Machine which runs on a mobile phone with limited resources, we could achieve frame rates of more than 12 fps. While this represents an important first step, further work is needed to fully implement and evaluate the framework.

Our main goal is to push forward the development of mobile phone-based smart camera networks and to make relevant code available to the public. Our vision is that in the future it will be possible to easily install the networks of smart cameras with little effort using by simply downloading the code onto the phone using 3G data networks. Our future work will focus on improving the system in order to enable concrete scenarios such as indoor navigation or object tracking. We also plan to publish the reference implementation of our software architecture as an open source project, and hope to establish a base for further community-driven development in the area. Further we want to add multiple view calibration in order to support high resolution object detection. As the vision capabilities of mobile phone become more elaborate, we hope to be able to work more on object recognition.

7. REFERENCES

[1] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury. A Survey on Wireless Multimedia Sensor Networks. *Computer Networks*, 51:921–960, 2007.

[2] P. Baker and Y. Aloimonos. Calibration of a multicamera network. In *Fourth workshop on Omnidirectional Vision and Camera Networks (Omnivis 2003)*, Madison, WI, USA, June 2003.

[3] S. Basagni, R. Bruno, G. Mambrini, and C. Petrioli. Comparative performance evaluation of scatternet formation protocols for networks of bluetooth devices. *Wirel. Netw.*, 10(2):197–213, 2004.

[4] M. Bramberger, A. Doblander, A. Maier, B. Rinner, and H. Schwabach. Distributed embedded smart cameras for surveillance applications. *Computer*, 39(2):68, 2006.

[5] M. Bramberger, B. Rinner, and H. Schabach. An embedded smart camera on a scalable heterogeneous multi-dsp system. In *European DSP Education and Research Symposium (EDERS) 2004*, Nov 2004.

[6] G. Cappellini, Y. P. Ivanenko, R. E. Poppele, and F. Lacquaniti. Motor patterns in human walking and running. *Journal of Neurophysiology*, 95:3426–3437, 2006.

[7] I. Downes, L. B. Rad, and H. Aghajan. Development of a mote for wireless image sensor networks. In *COGIS 2006*, Paris, France, March 2006.

[8] J. Ellis and M. Young. JSR 172: J2me web services 1.0. `http://jcp.org/en/jsr/detail?id=172`, October, 2003.

[9] A. O. Ercan, A. E. Gamal, and L. J. Guibas. Object tracking in the presence of occlusions via a camera network. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 509–518, New York, NY, USA, 2007. ACM Press.

[10] S. Funiak, C. Guestrin, M. Paskin, and R. Sukthankar. Distributed localization of networked cameras. In *Proceedings of the fifth international conference on Information processing in sensor networks*, Nashville, Tennessee, USA, 2006.

[11] I. Gartner. Gartner dataquest, March 2007.

[12] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[13] S. Hengstler and H. Aghajan. A smart camera mote architecture for distributed intelligent surveillance. In *ACM SenSys Workshop on Distributed Smart Cameras (DSC)*, Boulder, Colorado, USA, October 2006.

[14] T. Irnich and P. Stuckmann. Analytical performance evaluation of internet access over gprs and its comparison with simulation results. In *The 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, 2002.

[15] J. Jannotti and J. Mao. Distributed calibration of smart cameras. In *Workshop on Distributed Smart Cameras (DSC 2006)*, Boulder, CO, USA, October

2006.

[16] P. Lin. Channel allocation for gprs with buffering mechanisms. *Wirel. Netw.*, 9(5):431–441, 2003.

[17] O. Malik. Moore's law reconsidered. *CNNmoney.com Business 2.0 Magazine*, April 3 2007.

[18] J. Newman, M. Wagner, M. Bauer, A. MacWilliams, T. Pintaric, D. Beyer, D. Pustka, F. Strasser, D. Schmalstieg, and G. Klinker. Ubiquitous tracking for augmented reality. In *3rd IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2004)*, Arlington, VA, USA, 2-5 November 2004.

[19] A. Rahimi, B. Dunagan, and T. Darrell. Simultaneous calibration and tracking with a network of non-overlapping sensors. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2004)*, June 2004.

[20] M. Rahimi, R. Baer, O. I. Iroezi, J. C. G. amd Jay Warrior, D. Estrin, and M. Srivastava. Cyclops: In situ image sensing and interpretation in wireless sensor networks. In *Sensys 2005*, Boulder, CO, USA, November 2005.

[21] M. Ringwald and K. Römer. Practical time synchronization for bluetooth scatternets. In *Proceedings of the 4th International Conference on Broadband Communications, Networks, and Systems (BROADNETS 2007)*, May 2007.

[22] K. Römer. Time Synchronization in Ad Hoc Networks. In *MobiHoc 2001*, Long Beach, USA, Oct. 2001.

[23] C. J. Taylor and B. Shirmohammadi. Self localizing smart camera networks and their applications to 3d modeling. In *Workshop on Distributed Smart Cameras (DSC 2006)*, Boulder, CO, USA, October 2006.

[24] W. Wolf, B. Ozer, and T. Lv. Architectures for distributed smart cameras. In *IEEE International Conference on Multimedia & Expo*, Baltimore, Maryland, USA, July 2003.

[25] Connected Limited Device Configuration (CLDC); http://java.sun.com/products/cldc/index.jsp, JSR 30, JSR 139.

[26] JORAM: Java Open Reliable Asynchronous Messaging. `http://joram.objectweb.org`.

[27] JSR 259: Ad hoc networking api. `http://jcp.org/en/jsr/detail?id=259`, January, 2006.

[28] JSR 82: Java apis for bluetooth wireless technology. `http://jcp.org/en/jsr/detail?id=82`, September, 2005.

[29] The JXTA Java Micro Edition (MIDP/CLDC/CDC) Project. `https://jxta-jxme.dev.java.net`.