Hindawi Publishing Corporation
EURASIP Journal on Image and Video Processing
Volume 2008, Article ID 542808, 22 pages
doi:10.1155/2008/542808

Research Article

A Scalable Clustered Camera System for Multiple Object Tracking

Senem Velipasalar,[1] Jason Schlessman,[2] Cheng-Yao Chen,[2] Wayne H. Wolf,[3] and Jaswinder P. Singh[4]

[1] *Electrical Engineering Department, University of Nebraska-Lincoln, Lincoln, NE 68588, USA*
[2] *Electrical Engineering Department, Princeton University, Princeton, NJ 08544, USA*
[3] *School of Electrical and Computer Engineering, Georgia Institute of Technology, GA 30332, USA*
[4] *Computer Science Department, Princeton University, Princeton, NJ 08544, USA*

Correspondence should be addressed to Senem Velipasalar, velipasa@engr.unl.edu

Received 1 November 2007; Revised 21 March 2008; Accepted 12 June 2008

Recommended by Andrea Cavallaro

Reliable and efficient tracking of objects by multiple cameras is an important and challenging problem, which finds wide-ranging application areas. Most existing systems assume that data from multiple cameras is processed on a single processing unit or by a centralized server. However, these approaches are neither scalable nor fault tolerant. We propose multicamera algorithms that operate on *peer-to-peer* computing systems. Peer-to-peer vision systems require codesign of image processing and distributed computing algorithms as well as sophisticated communication protocols, which should be carefully designed and verified to avoid deadlocks and other problems. This paper introduces the scalable clustered camera system, which is a *peer-to-peer* multicamera system for multiple object tracking. Instead of transferring control of tracking jobs from one camera to another, each camera in the presented system performs its own tracking, keeping its own trajectories for each target object, which provides fault tolerance. A fast and robust tracking algorithm is proposed to perform tracking on each camera view, while maintaining consistent labeling. In addition, a novel communication protocol is introduced, which can handle the problems caused by communication delays and different processor loads and speeds, and incorporates variable synchronization capabilities, so as to allow flexibility with accuracy tradeoffs. This protocol was exhaustively verified by using the *SPIN* verification tool. The success of the proposed system is demonstrated on different scenarios captured by multiple cameras placed in different setups. Also, simulation and verification results for the protocol are presented.

1. INTRODUCTION

This paper takes a holistic view of the problems that need to be solved to build scalable multicamera systems. Scalability has two aspects: computation and communication. In order to address scalability in both of these aspects, we present the scalable clustered camera system (SCCS), which is a *distributed smart camera* system for multiobject tracking. In a *smart camera* system, each camera is attached to a computing component, in this case different CPUs. In other words, in SCCS, a different processing unit is used to process each camera, which provides scalability in computation power. Moreover, processing units communicate with each other in a peer-to-peer fashion eliminating the need for a central server, and thus providing scalability on the communication side and also removing a single point of failure.

Reliable and efficient tracking of objects by multiple cameras is an important and challenging problem, which finds wide-ranging application areas such as video surveillance, indexing and compression, gathering statistics from videos, traffic flow monitoring, and smart rooms. Due to the inherent limitations of a single visual sensor, such as limited field of view and delays due to panning and tilting, using multiple cameras has become inevitable for object tracking in applications requiring increased coverage of areas. Multiple cameras can enhance the capability of vision applications, providing fault-tolerance and robustness for issues such as target occlusion. Hence, many research groups have studied multicamera systems [1–30].

Yet, using multiple cameras to track multiple objects poses additional challenges. One of these challenges is

the *consistent labeling* problem, that is, establishing correspondences between moving objects in different views. Multicamera systems, rather than treating each camera individually, compare features and trajectories from different cameras in order to obtain history of the object movements, and handle the loss of the target objects which may be caused by occlusion or errors in the background subtraction (BGS) algorithms.

Different approaches have been taken to solve the consistent labeling problem. Kelly et al. [14] assume that all cameras are fully calibrated. Funiak et al. [31] propose a distributed calibration algorithm for camera networks. Chang and Gong [8] and Utsumi et al. [25] employ feature matching. Yet, relying on feature matching can cause problems as the features can be seen differently by different cameras due to lighting variations. In addition, calibrating cameras fully is expensive and impractical to install by the end user, since it requires some expert intervention.

Kettnaker and Zabih [15] use observation intervals and transition times of objects across cameras for tracking. Lee et al. [18] assume that the intrinsic camera parameters are known and use the centroids of the tracked objects to estimate a homography and to align the scene ground plane across multiple views. Khan and Shah [16] present a method that uses *field of view* (FOV) lines and does not require the cameras to be calibrated. However, due to the way the lines are recovered, they may not be localized reliably, and if there is dense traffic around the FOV line, the method can result in inconsistent labels. More recently, Calderara et al. [6, 7] and Hu et al. [12] presented methods for consistent labeling making use of the principle axes of people.

Cai and Aggarwal [5] present a system in which only the neighboring cameras are calibrated to their relative coordinates. Targets are tracked using a single camera view until the system predicts that the active camera will no longer have a good view of the object. The tracking algorithm then switches to another camera. Nguyen et al. [22] introduce a system in which all cameras are calibrated, and the most appropriate camera is chosen to track each object. The tracking job is assigned to the camera that is closest to the object. However, with a handoff or switching approach, even though increased coverage is provided, and objects can be tracked for longer periods of time, the tracking is still being performed on one camera view at a time. This is not a fault-tolerant approach, since the camera responsible for tracking can break down, and detecting and recovering from this, if recovery is possible, can introduce some delay. In addition, the camera assigned for the tracking of an object may not have the view of the object we are interested in. For instance, the camera closest to a person may be viewing that person from the back while the interest is seeing that person's face. Moreover, occlusions, merging/splitting of objects, foreground blobs that are detected incompletely may cause frequent switching of cameras.

In order to provide fault-tolerance, robustness, and multiple views of the tracked targets, objects in the overlapping regions of the fields of view of cameras should be tracked simultaneously on those views with consistent labels. This way, even if a camera breaks down, other cameras can still continue tracking and trajectories of the same object from different views can be obtained. Also, in the case of occlusion or loss of target objects, data about the object features and trajectories in different camera views can be exchanged to keep the trajectories updated in all camera views.

As cameras become less expensive, many systems will use large numbers of cameras for better coverage and higher accuracy. Thus, scalability and computational efficiency of multicamera systems are very important issues that need to be addressed. The performance and scalability of such systems should not be debilitated with additional cameras. Although many groups have developed methods to combine data from multiple cameras, much less attention has been paid to the computational efficiency and scalability. Many existing systems assume that multiple cameras are processed on a single CPU or by a centralized server. However, these are not scalable approaches, and they introduce a single point of failure. Chang and Gong [8] propose a system that is implemented on an SGI workstation with two cameras. For a single CPU system, the amount of processing necessary to track multiple objects on multiple views can be excessive for real-time performance. Moreover, scalability is debilitated as each additional camera imposes greater performance requirements.

In order to increase processing power, and handle multiple video streams, *distributed systems* have been employed instead of using a single CPU. In a distributed system, different CPUs are used to process inputs from different cameras. Yet, most existing distributed multicamera systems use a centralized server/control center. Yuan et al. [30] present a distributed surveillance system in which computers are connected to a server, and camera units do not collaborate with each other or exchange information. Collins et al. [9, 10] introduced the VSAM system, where all resulting object hypotheses from all sensors are transmitted at every frame back to a central operator control unit. Nguyen et al. [20] propose a multicamera system where all the local processing results are sent to a main controller. Lo et al. [32] introduce a multisensor distributed system where a central server coordinates the processing of the sensor inputs. Krumm et al. [17] present a multicamera multiperson tracking system for the *EasyLiving* project. They use two sets of color stereo cameras, each connected to its own computer. A program called stereo module locates people-shaped blobs, and reports the $2D$ ground plane locations of these blobs to a tracking program running on a third computer. Using a central server or a control unit for data coordination/integration simplifies some problems, such as video synchronization and communication between the algorithms handling the various cameras. But server-based multicamera systems have a bandwidth scaling problem, since the central server can quickly become overloaded with the aggregate sum of messages/requests from an increased number of nodes. Also, the server is a single point of failure for the whole system. In addition, server-based systems are not practical in many realistic environments, and have high installation costs. Besides the algorithm development, hardware design and resource management have also been considered for parallel processing. Watlington and Bove [33]

propose a data-flow model and use a distributed resource manager to support parallelism for media processing.

The aforementioned problems of server-based systems necessitate the use of *peer-to-peer* systems, where individual nodes communicate with each other without going through a centralized server. Several important issues need to be addressed when designing peer-to-peer systems. First, communication between processing elements takes a significant amount of time. This necessitates the design of tracking algorithms requiring relatively little interprocess communication between the nodes. Decreasing the number of messages between the nodes also requires a careful design and choice of when to trigger the data transfer, what data to send in what fashion, and to whom to send this data. The system must also maintain the consistency of data across nodes as well as operations upon the data without use of a centralized server. Also, even if the cameras and input video sequences are synchronized, communication and processing delays pose a serious problem. Depending on the amount of processing each processor has to do, one processor can run faster/slower than the other. Thus, when a processor receives a request, it may be ahead/behind compared to the requester. These issues mandate efficient and sophisticated communication protocols for peer-to-peer systems.

Atsushi et al. [1] use multiple cameras attached to different PCs connected to a network. They calibrate the cameras to track objects in world-coordinates, and send message packets between stations. Ellis [11] also uses a network of calibrated cameras. Bramberger et al. [3] present a distributed embedded smart camera system with loosely coupled cameras. They use predefined migration regions to handover the tracking process from one camera to another. But, these methods do not discuss the type and details of communication, and how to address the communication and processing delay issues.

As stated previously, peer-to-peer systems require efficient and sophisticated communication protocols. These protocols find use in real-time systems, which tend to have stringent requirements for proper system functionality. Hence, the protocol design for these systems necessitates transcending typical qualitative analysis using simulation and instead, requires verification. The protocol must be checked to ensure that it does not cause unacceptable issues such as deadlocks and process starvation, and has correctness properties such as the system eventually reaching specified operating states.

Verification of communication protocols is a rich topic, particularly for security and cryptographic systems. Karlof et al. [34] analyzed the security properties of two cryptographic protocols. Evans and Schneider [35] verified time-dependent authentication properties of security protocols. Vanackère [36] modeled cryptographic protocols as a finite number of processes interacting with a hostile environment, and proposed a protocol analyzer trust for verification. Finally, a burgeoning body of work exists pertaining to the formal verification of networked multimedia systems. Bowman et al. [37] described multimedia stream as a timed automata, and verified the satisfaction of quality of service (*QoS*) properties including throughput and end-to-end latency. Sun et al. [38] proposed a testing method for verifying QoS functions in distributed multimedia systems, where media streams are modeled as a set of timed automata.

Our previous work [27, 28] introduced some of the tools necessary towards building a peer-to-peer camera system. The work presented in [28] performs multicamera tracking and information exchange between cameras. However, it was implemented on a single CPU in a sequential manner, and the tracking algorithm used required more data transfer. This paper presents SCCS together with its communication protocol and its exhaustive verification results. SCCS is a scalable peer-to-peer multicamera system for multiobject tracking. It is a *smart camera* system wherein each camera is attached to a computing component, in this case different CPUs. The peer-to-peer communication protocol is designed so that the number of messages that are sent between the nodes is decreased, and the *system synchronization* issue is addressed.

A computationally efficient and robust tracking algorithm is presented to perform tracking on each camera view, while maintaining consistent labeling. Instead of transferring control of tracking jobs from one camera to another, each camera in SCCS performs its own tracking and keeps its own trajectories for each target object, thus providing fault tolerance. Cameras can communicate with each other to resolve partial/complete occlusions, and to maintain consistent labeling. In addition, if the location of an object cannot be determined at some frame reliably due to the errors resulted from BGS, the trajectory of that object is robustly updated from other cameras. SCCS keeps trajectories updated in all views without any need for an estimation of the moving speed and direction, even if the object is totally invisible to that camera. Our tracking algorithm deals with the cases of merging/splitting on a single camera view without sending requests to other nodes in the system. Thus, it provides coarse object localization with sparse message traffic.

In addition, we introduce a novel communication protocol that coordinates multiple tracking components across the distributed system, and handles the processing and communication delay issues. The decisions about when and with whom to communicate are made such that the frequency and size of transmitted messages are kept small. This protocol incorporates variable synchronization capabilities, so as to allow flexibility with accuracy tradeoffs. Nonblocking sends and receives are used for message communication, since for each camera it is not possible to predict when and how many messages will be received from other cameras. Moreover, the type of data that is transferred between the nodes can be changed, depending on the application and what is available, and our protocol remains valid and can still be employed. For instance, when full calibration of all the cameras is tolerated, the $3D$ world coordinates of the objects can be transferred between the nodes. We exhaustively verified this protocol with success by using the *SPIN* verification tool.

We introduced the initial version of the SCCS and its communication protocol in [29, 39]. We extended the communication protocol to address real-time concerns, and to handle conflicts in received replies. In this paper, we

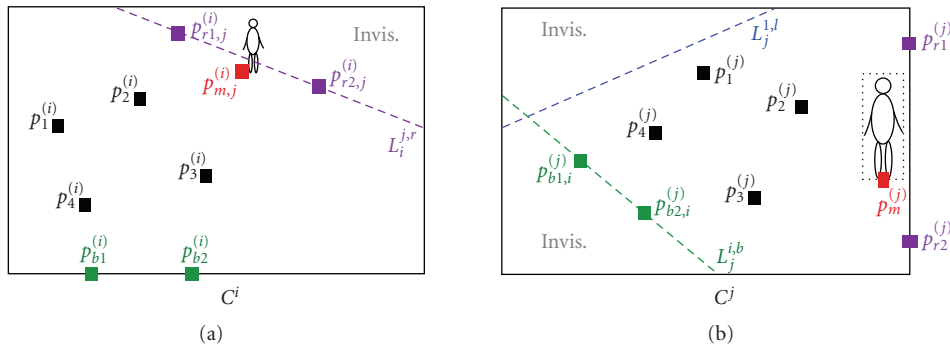


FIGURE 1: $p^{(j)}_{b1,i}$ and $p^{(j)}_{b2,i}$ are the corresponding locations of $p^{(i)}_{b1}$ and $p^{(i)}_{b2}$, respectively, and the recovered FOV line passing through $p^{(j)}_{b1,i}$ and $p^{(j)}_{b2,i}$ is shown with a dashed green line on the view of C^j.

present this protocol and its verification results in detail. We also compare the number of messages that need to be sent around in a server-based scenario and in our system, and present results of this comparison. We show that, contrary to the server-based scenario, the total number of messages sent around by our system is independent of the number of trackers in each camera view. In addition, we compare the server-based system scenario with our peer-to-peer system in terms of the message *loads* on the individual nodes, and show that the number of messages a single node has to handle is considerably less in our peer-to-peer system. We present results of different sets of experiments that were performed to obtain the speed up provided by SCCS, to measure average data transfer accuracy and average *waiting time*. Experimental results demonstrate the success of the proposed peer-to-peer multicamera tracking system, with a minimum accuracy of 94.2% and 90% for *new_label* and *lost_label* cases, respectively, with a high frequency of synchronization. We also present the results obtained after exhaustively verifying the presented communication protocol with different communication scenarios.

The rest of the paper is organized as follows: Section 2 describes the computer vision algorithms in general. More specifically, the recovery of FOV lines, the consistent labeling algorithm, and the tracking algorithm are described in Sections 2.1.1, 2.2, and 2.3, respectively. The communication protocol is introduced in Section 3, and its verification and obtained results are described in Section 4. Section 5 presents the experimental results obtained with several different video sequences with varying difficulty, and Section 6 concludes the paper.

2. MULTICAMERA MULTIOBJECT TRACKING

2.1. Field of view (FOV) lines

Khan and Shah [16] introduced the FOV lines, and showed that when FOV lines are recovered, the consistent labeling problem can be solved successfully. The 3D FOV lines of camera C^i are denoted by $L^{i,s}$ [16], where $s \in \{r,l,t,b\}$ correspond to one of the sides of the image plane. The projections of the 3D FOV lines of camera C^i onto the image plane of camera C^j will result in 2D lines denoted by $L^{i,s}_j$, and called the *FOV lines*.

2.1.1. Recovery of field of view lines

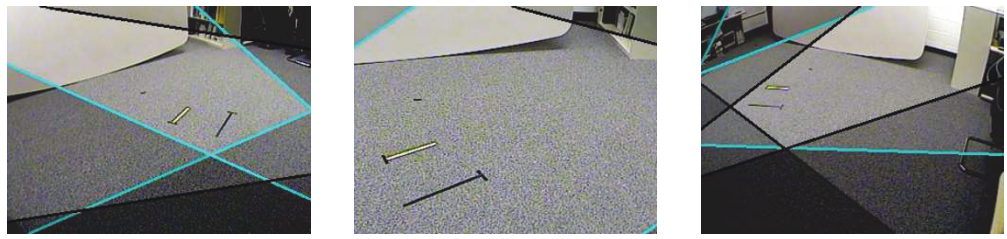
Khan and Shah [16] recover FOV lines by observing moving objects in different views and using entry/exit events. However, since the method relies on object movement in the environment, there needs to be enough traffic across a particular FOV line to be able to recover it. In addition, the output of this method can be affected by the performance of the BGS algorithm. Depending on the size of the objects, they may not be detected instantly or entirely, and thus, FOV lines may not be located reliably.

Since FOV lines will play an important role in our consistent labeling algorithm and also in our communication protocol later, it is necessary to recover all of them in a robust way. We present robust and reliable methods for recovering FOV lines and for consistent labeling. The method presented for the recovery of FOV lines does not rely on the object movement in the scene or on the performance of BGS algorithms. This way, all visible FOV lines in a camera view can be recovered at the beginning even if there is no traffic at the corresponding region. In addition, there is no need to know the intrinsic or extrinsic camera parameters. It is assumed that cameras have overlapping fields of view, and the scene ground is planar. Then a homography is estimated to recover the FOV lines.

The inputs to the proposed system are four pairs of corresponding points (chosen offline on the ground plane) on two camera views. These points in the views of C^i and C^j are denoted by $P^{(i)} = \{p^{(i)}_1,\ldots,p^{(i)}_4\}$ and $P^{(j)} = \{p^{(j)}_1,\ldots,p^{(j)}_4\}$, respectively, and are displayed in black in Figure 1. Let $\vec{p}^{(i)}_k = (x^{(i)}_k, y^{(i)}_k, 1)^T$, where $k \in \{1,\ldots,4\}$, denote the homogeneous coordinates of the input point $p^{(i)}_k = (x^{(i)}_k, y^{(i)}_k)$. Then, a homography (H) is estimated from $\{\vec{p}^{(i)}_1,\ldots,\vec{p}^{(i)}_4\}$ and $\{\vec{p}^{(j)}_1,\ldots,\vec{p}^{(j)}_4\}$ by using the direct linear transformation algorithm described by Hartley and Zisserman [40].

(a) Setup 1—Camera 1 (b) Setup 1—Camera 2 (c) Setup 2—Camera 1 (d) Setup 2—Camera 2

Figure 2: (a)-(b) and (c)-(d) show the recovered FOV lines for two different camera setups. The shaded regions are outside the FOV of the other camera.



(a) (b) (c)

Figure 3: (a), (b), and (c) show the recovered FOV lines. The shaded regions are outside the FOV of the other cameras.

The image of the camera view whose FOV lines will be recovered on the other view is called the *field image*. After the homography is estimated, the system finds two points on one of the boundaries of the *field image*, so that each of them is in general position with the four input points. Then it checks with the user that these boundary points are coplanar with the four input points. Let us denote the two points found on the image boundary s of the camera C^i by $p_{s_1}^{(i)} = (x_{s_1}^{(i)}, y_{s_1}^{(i)})$ and $p_{s_2}^{(i)} = (x_{s_2}^{(i)}, y_{s_2}^{(i)})$, where $s \in \{r, l, t, b\}$ correspond to one of the sides of the image plane (Figure 1). The corresponding locations of $(x_{s_1}^{(i)}, y_{s_1}^{(i)})$ and $(x_{s_2}^{(i)}, y_{s_2}^{(i)})$ on the view of camera C^j are denoted by $p_{s_1,i}^{(j)} = (x_{s_1,i}^{(j)}, y_{s_1,i}^{(j)})$ and $p_{s_2,i}^{(j)} = (x_{s_2,i}^{(j)}, y_{s_2,i}^{(j)})$, and computed by using

$$H\vec{p}_{s_n}^{(i)} \cong \vec{p}_{s_n,i}^{(j)}, \tag{1}$$

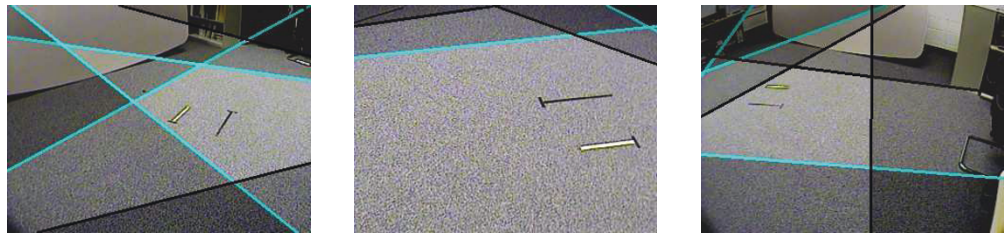
where $n \in \{1, 2\}$, $\vec{p}_{s_n}^{(i)} = (x_{s_n}^{(i)}, y_{s_n}^{(i)}, 1)^T$, and $\vec{p}_{s_n,i}^{(j)}$ denotes the homogeneous coordinates of $p_{s_n,i}^{(j)} = (x_{s_n,i}^{(j)}, y_{s_n,i}^{(j)})$. $x_{s_n,i}^{(j)}$ and $y_{s_n,i}^{(j)}$ are obtained by normalizing $\vec{p}_{s_n,i}^{(j)}$, so that its third entry is equal to 1. Once $p_{s_1,i}^{(j)}$ and $p_{s_2,i}^{(j)}$ are obtained on the other view, the line going through these points defines the FOV line corresponding to the image boundary s of the camera C^i. Two points are found on each boundary of interest and the FOV line corresponding to that boundary is recovered similarly. Let us illustrate these steps by an example referring to Figure 1. Let the boundary of interest be the bottom boundary of the view of C^i, thus side s is b. The system finds $p_{b_1}^{(i)}$ and $p_{b_2}^{(i)}$ on this boundary, which are displayed in green in Figure 1(a), as described above. Then, the corresponding locations of these points on the view of C^j are computed by using (1). These corresponding points are denoted by $p_{b_1,i}^{(j)}$ and $p_{b_2,i}^{(j)}$, and are displayed in green in Figure 1(b). The line going through $p_{b_1,i}^{(j)}$ and $p_{b_2,i}^{(j)}$ is the FOV line corresponding to the bottom boundary of camera C^i. This line is denoted by $L_j^{i,b}$, and is shown with a dashed green line on the view of C^j in Figure 1(b).

Figures 2 to 4 show the recovered FOV lines for different video sequences and camera setups. Although there was no traffic along the right boundary of Figure 2(b), the corresponding FOV line is successfully recovered as shown in Figure 2(a).

2.1.2. Checking object visibility

As stated previously, $L_j^{i,s}$ denotes the projection of the 3D FOV line $L^{i,s}$ onto the view of C^j and is represented by the equation of the line, which is written as $y = Sx + C$. Henceforth, a point $p_m^{(j)} = (x_m^{(j)}, y_m^{(j)})$ will be considered on the visible side of $L_j^{i,s}$ if $\text{sign}(y_m^{(j)} - Sx_m^{(j)} - C) = \text{sign}(y_a^{(j)} - Sx_a^{(j)} - C)$, where $(x_a^{(j)}, y_a^{(j)})$ are the coordinates of the $p_a^{(j)}$ which is any one of the input points in $P^{(j)}$.

When an object $O^{(j)}$ enters the view of C^j, BGS is applied first and a bounding box around the foreground (FG) object is obtained. Then, its visibility in the view of C^i is checked by employing $L_j^{i,s}$. The midpoint $(p_m^{(j)})$ of the bottom line of the bounding box of the object is used as its location. If this point lies on the visible side of $L_j^{i,s}$, for all $s \in \{r, l, t, b\}$, then it is deduced that $O^{(j)}$ is visible by C^i (Figure 1).



(a) (b) (c)

Figure 4: (a), (b), and (c) show the recovered FOV lines. The shaded regions are outside the FOV of the other cameras.

2.2. Consistent labeling

The consistent labeling scheme of Khan and Shah [16] is based on the minimum distance between FG objects and a FOV line. However, performing the labeling based only on the FOV lines can be error-prone as more than one object can be in the vicinity of the line especially in crowded scenes, or multiple people can enter one view at the same time. Also, if the entry of an object is detected with a delay due to the errors in BGS, the corresponding object in the other camera will move further away from the line which may cause another object with the wrong label to be closer to the FOV line.

Instead of relying only on the FOV lines, the following steps are employed for more robustness: first, the visibility of a new object $O^{(j)}$, entering the view of C^j from side s, by C^i is checked as described in Section 2.1.2. Then, the corresponding location $(p^{(i)}_{m,j})$ of $p^{(j)}_m$ in the view of C^i is calculated by using (1) as depicted in Figure 1. The foreground objects in the view of C^i, which were on the invisible side of $L^{j,s}_i$ and move in the direction of the visible side of the line, are determined. From those foreground objects, the one that is closest to the point $p^{(i)}_{m,j}$ is found, and its label is given to the $O^{(j)}$. Similarly, if multiple people enter the scene simultaneously, the algorithm uses their calculated corresponding locations in the other view to resolve the ambiguity.

2.3. The tracking algorithm: coarse object localization with sparse message traffic

Our proposed tracking algorithm allows for sparse message traffic by handling the cases of merging and splitting within a single camera view without sending request messages to other cameras.

First, FG objects are segmented from the background in each camera view by using the BGS algorithm presented by Stauffer and Grimson [41], which employs adaptive background mixture models. Then, connected component analysis is performed, which results in FG blobs. When a new FG blob is detected within the camera view, a new tracker is created, and a mask for the tracker is built where the FG pixels from this blob and background pixels are set to be 1 and 0, respectively. The box surrounding the FG pixels of the mask is called the *bounding box*. Then, the color histogram of the blob is learned from the input image, and is saved as the *model histogram* of the tracker.

At each frame, the trackers are matched to detected FG blobs by using a computationally efficient blob tracker which uses a matching criteria based on the bounding box intersection and the Bhattacharya coefficient $\rho(\mathbf{y})$ [42] defined by

$$\rho(\mathbf{y}) \equiv \rho[p(\mathbf{y}), q] = \int \sqrt{p_{\mathbf{z}}(\mathbf{y}) q_{\mathbf{z}}}\, d\mathbf{z}. \tag{2}$$

In (2), $\mathbf{z}$ is the feature representing the color of the target model and is assumed to have a density function $q_{\mathbf{z}}$ while $p_{\mathbf{z}}(\mathbf{y})$ represents the color distribution of the candidate FG blob centered at location $\mathbf{y}$. The Bhattacharya coefficient is derived from the sample data by using

$$\hat{\rho}(y) \equiv \rho[\hat{\mathbf{p}}(\mathbf{y}), \hat{\mathbf{q}}] = \sum_{u=1}^{m} \sqrt{\hat{p}_u(\mathbf{y}) \hat{q}_u}, \tag{3}$$

where $\hat{\mathbf{q}} = \{\hat{q}_u\}_{u=1\ldots m}$, and $\hat{\mathbf{p}}(\mathbf{y}) = \{\hat{p}_u(\mathbf{y})\}_{u=1\ldots m}$ are the probabilities estimated from the m-bin histogram of the model and the candidate blobs, respectively. These probabilities are estimated by using the color information at the nonzero pixel locations of the masks. If the bounding box of an FG blob intersects with that of the current model mask of the tracker, the Bhattacharya coefficient between the model histogram of the tracker and the color histogram of the FG blob is calculated by using (3). The tracker is assigned to the FG blob which results in the highest Bhattacharya coefficient, and the mask, and thus the bounding box, of the tracker are updated. The Bhattacharya coefficient with which the tracker is matched to its object is called the *similarity coefficient*. If the similarity coefficient is greater than a predefined distribution update threshold, the model histogram of the tracker is updated to be the color histogram of the FG blob to which it is matched.

Based on this matching criteria, when objects merge, multiple trackers are matched to one FG blob, and the labels of all matched trackers are displayed on this blob, as shown in Figures 5, 6, 23, and 24. The masks of the trackers are then updated in the previously discussed fashion. The trackers that are matched to the same FG blob are put into a *merge state*, and in this state their model histograms are *not* updated. When objects split from each other, trackers

(a) Frame 530 (a′) Frame 530 (b) Frame 570 (b′) Frame 570 (c) Frame 590 (c′) Frame 590

FIGURE 5: Example of successfully resolving a merge. (a), (b), (c), (a′), (b′), and (c′) show the original images, and the tracked objects with their labels, respectively.

(a) Frame 795 (b) Frame 815 (c) Frame 877 (d) Frame 920 (e) Frame 950

FIGURE 6: Example of resolving the merging of multiple objects.

are matched to their targets based on the bounding box intersection and Bhattacharya coefficient criteria mentioned above.

There may be rare but unfavorable cases where an FG object, appearing after the split of merged objects, may not be matched to its tracker. We deal with these cases as follows: denote two trackers by T_1 and T_2, and their target objects by O_1 and O_2, respectively. When these objects merge, $O_{1\cup2}$ is formed, and T_1 and T_2 are both matched to $O_{1\cup2}$. After O_1 and O_2 split, $B_{T_iO_j}$ are calculated, where $\{i, j\} \in \{1, 2\}$, and $B_{T_iO_j}$ denotes the Bhattacharya coefficient calculated between the histograms of T_i and O_j. Based on $B_{T_iO_j}$, both T_1 and T_2 can still be matched to O_2, for instance, and stay in the *merge state*. Denote the similarity coefficient of T_i by S_{T_i}. Thus, in this case, $S_{T_1} = B_{12}$ and $S_{T_2} = B_{22}$. This can happen because the model distributions of the trackers are not updated during the *merge state*, and there may be changes in the color of O_1 during and after merging. Another reason may be O_1 and O_2 having similar colors from the outset. When this occurs, O_1 is compared against the trackers which are in the *merge state* and intersect with the bounding box of O_1. That is, it is compared against T_1 and T_2, and $B_{T_1O_1}$ and $B_{T_2O_1}$ are calculated. Then, O_1 is assigned to the tracker T_{i*}, where

$$i^* = \underset{i\in\{1,2\}}{\operatorname{argmin}} \left(S_{T_i} - B_{T_iO_1}\right). \tag{4}$$

If a foreground blob cannot be matched to any of the trackers, and if there are trackers in the *merge state*, the unmatched object is compared against those trackers by using the logic in (4), which is also applicable if there are more than two trackers in the *merge state* as shown in Figures 6 and 23.

As stated previously, this algorithm provides coarser object localization and decreases the message traffic by not sending a request message each time a merging or splitting occurs. *If* the exact location of an object in the blob, formed after the merging, is required, we propose another algorithm that can be used at the expense of more message traffic: when a tracker is in the *merge state*, other nodes that can see its most recent location can be determined as described in Section 2.1.2, and a request message can be sent to these nodes to retrieve the location of the tracker in the *merge state*. If the current location of the tracker is not visible by any of the other cameras, then the mean-shift tracking [42] can be activated.

The mean-shift tracking is error-prone since it can be distracted by the background. It is also computationally more expensive. Thus, when a tracker is in the *merge state*, it is preferable to send messages to other nodes, and request the location of this tracker, if its most recent location is in their FOV. Thus, this algorithm requires additional message traffic. We proposed this second algorithm as an alternative if the *exact* location of each tracker in the *merge state* is required. The experiments presented in Section 5 were performed by using the first proposed algorithm as the tracking component of the SCCS.

3. INTER-CAMERA COMMUNICATION PROTOCOL

One issue that needs to be addressed when using peer-to-peer systems is that communication is expensive and takes a significant amount of time. Also, the number of messages that are sent between the nodes should be decreased to save power and increase speed. Another issue is maintaining consistency for data across cameras as well as operations upon the data without the use of a centralized server. Even if the cameras and input video sequences are synchronized, communication and processing delays pose a serious problem. The processors will have different amounts of processing to do, and may also run at different processing rates. These, coupled with potential network delays, cause one processor to be ahead of/ behind the others during execution. Thus, when a processor receives a request, it may be ahead/behind compared to the requester. Hence, *system synchronization* becomes very important to ensure the transfer of coherent vision data between cameras. All these issues mandate an efficient and sophisticated communication protocol.

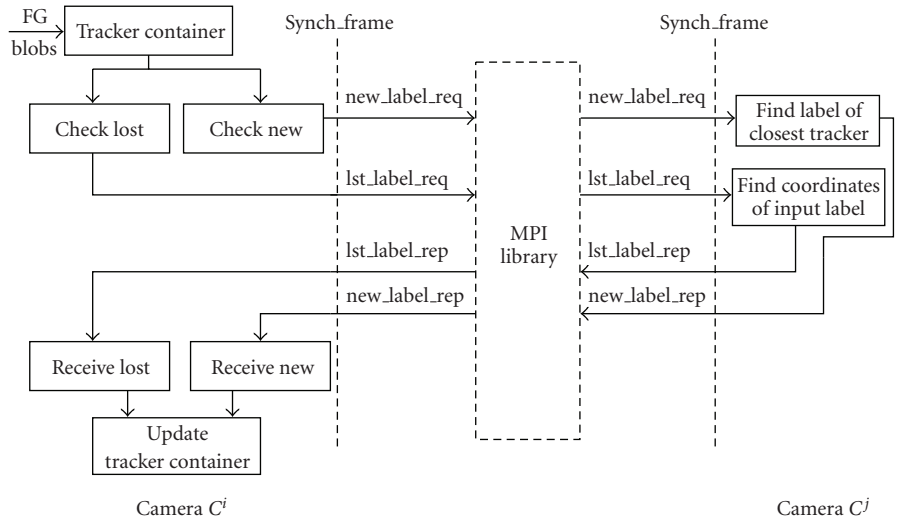


Figure 7: Communication between two cameras.

The protocol of SCCS utilizes point-to-point communication, as opposed to some previous approaches that require a central message processing server. Our approach offers a latency advantage and removes the single point of failure. Moreover, nodes do not need to send the state of the trackers to a server at every single frame. Thus, contrary to the server-based scenario, the total number of messages sent around by our system is independent of the number of trackers in each camera view. This decreases the number of messages considerably as will be discussed in Section 3.5. Furthermore, compared to server-based systems, the proposed protocol decreases the message load on each node. This design is more scalable, since for a central server implementation, the server quickly becomes overloaded with the aggregate sum of messages and requests from an increased number of nodes.

In this section, the protocol of SCCS is presented, which is a novel peer-to-peer communication protocol that can handle communication and processing delays and hence maintain consistent data transfer across multiple cameras. This protocol is designed by determining the answers to the following questions.

(a) *When to communicate*: determining the events which will require the transfer of data from other cameras. These events will henceforth be referred to as *request events*.

(b) *With whom to communicate*: determining the cameras to which requests should be sent.

(c) *What to communicate*: choosing the data to be transferred between the cameras.

(d) *How to communicate*: designing the manner in which the messages are sent, and determining the points during execution at which data transfers should be made.

The protocol is designed so that the number of messages that are sent between the nodes is decreased, and the *system synchronization* issue is addressed.

The block diagram in Figure 7 illustrates the concepts discussed in this section. It should be noted that, at some point during execution, each camera node can act as the requesting or replying node. The implementation of the proposed system consists of a parallel computing cluster with communication between the nodes performed by the message passing interface (MPI) library [43]. In this work, the use of MPI is illustrative but not mandatory since it, like other libraries, provides well-defined communication operations including blocking/nonblocking send and receive, broadcast, and gathering. MPI is also well defined for inter- and intra-group communication and can be used to manage large camera groups. We take advantage of the proven usefulness of this library, and treat it as a transparent interface between the camera nodes.

3.1. When to communicate: request events

A camera will need information from the other cameras when (a) a new object appears in its FOV, or (b) a tracker cannot be matched to its target object. These events are called *request events*, and are referred to as *new_label* and *lost_label* events, respectively. If one of these events occurs within a camera's FOV, the processor processing that camera needs to communicate with the other processors.

In the *new_label* case, when a new object is detected in the current camera view, it is possible that this object was already being tracked by other cameras. If this is the case, the camera will issue a *new_label* request to those cameras to receive the existing label of this object, and to maintain consistent labeling.

Camera C^i could also need information from another node when a tracker in C^i cannot be matched to its target object, and this is called the *lost_label* case. This may occur, for instance, if the target object is occluded in the scene or cannot be detected as an FG object at some frame due to the failure of the BGS algorithm. In this case, a *lost_label* request

will be sent to the appropriate node to retrieve and update the object location.

Another scenario where communication between the cameras may become necessary is when trackers are merged and the location of each merged object is required. However, if the exact location of the object is not required, and coarser localization is tolerated, then the tracking algorithm introduced in Section 2.3 can be used to handle the merging/splitting within single camera view without sending request messages to the other nodes.

3.2. With whom to communicate

The proposed protocol is designed such that rather than sending requests to every single node in the system, requests are sent to the processors who can provide the answers for them. This is achieved by employing the FOV lines.

When a request needs to be made for an object $O^{(j)}$ in the view of C^j, the visibility of this object by camera C^i is checked using the FOV lines as described in Section 2.1.2. If it is deduced that the object is visible by C^i, a request message targeted for node i is created and the ID of the target processor, which is i in this case, is inserted into this message. Similarly, a list of messages for all the cameras that can see this object is created.

3.3. What to communicate

The protocol sends minimal amounts of data between different nodes. Messages consist of 256-byte packets, with character command tags, integers, and floats for track labels and coordinates, respectively, and integers for camera ID numbers. Clearly, this is significantly less than the amount of data inherent in transferring streams of video or even image data and features. Considering that integers and floats are 4 bytes, and a character is 1 byte, we currently do not use all of the 256 bytes. As more features are discovered that are useful and important to transfer between cameras, they will be inserted into the message packets. Messages that are sent between the processors are classified into four categories: (1) new label request messages, (2) lost label request messages, (3) new label reply messages, and (4) lost label reply messages.

3.3.1. New label request case

If an FG object viewed by camera C^i cannot be matched to any existing tracker, a new tracker is created for it, all the cameras that can see this object are found by using the FOV lines, and a list of cameras to communicate is formed. A request message is created to be sent to the cameras in this list. The format of this message is

$$Cmd\_tag \quad Target\_id \quad Curr\_id \quad Side \quad x \quad y \quad Curr\_label. \tag{5}$$

In this case, *Cmd_tag* is a character array that holds *NEW_LABEL_REQ* indicating that this is a request message for the *new_label* case. *Target_id* and *Curr_id* are integers. *Target_id* is the ID of the node to which this message is addressed, and *Curr_id* is the ID of the node that processes the input of the camera which needs the label information. For instance, *Curr_id* is i in this case. These ID numbers are assigned to the nodes by MPI at the beginning of the execution. *Side* is a character array that holds information about the side of the image from which the object entered the scene. Thus, it can be *right, left, top, bottom,* or *middle*. The next two entities, x and y, are floats representing the coordinates of the location $(p_m^{(i)})$ of the object in the coordinate system of C^i. Finally, *Curr_label* is an integer holding the temporary label given to this object by C^i. The importance and benefit of using this temporary label will be clarified in Sections 3.4 and 5.

3.3.2. Lost label request case

For every tracker that cannot find its match in the current frame, the cameras that can see the most recent location of its object are determined by using FOV lines. Then, a *lost_label* request message is created to be sent to the appropriate nodes to retrieve the updated object location. The format of a *lost_label* message is

$$Cmd\_tag \quad Target\_id \quad Curr\_id \quad Lost\_label \quad x \quad y. \tag{6}$$

Cmd_tag is a character array that holds *LOST_LABEL_REQ* indicating that this is a request message for the *lost_label* case. *Target_id* and *Curr_id* are the same as described above. *Lost_label* is an integer that holds the label of the tracker, which could not be matched to its target object. Finally, x and y are floats that are the coordinates of the latest location $(p_m^{(i)})$ of the tracker in the coordinate system of C^i.

3.3.3. New label reply case

If node j receives a message, and the *Cmd_tag* of this message holds *NEW_LABEL_REQ*, then node j needs to send back a reply message. The format of this message is

$$Cmd\_tag \quad Temp\_label \quad Answer\_label \quad Min\_pnt\_dist. \tag{7}$$

In this case, *Cmd_tag* is a string that holds *NEW_LABEL_REP* indicating that this is a reply message to a *new_label* request. *Temp_label* and *Answer_label* are integers. *Temp_label* is the temporary label given to a new object by the requesting camera, and *Answer_label* is the label given to the same object by the replying camera. Finally, *Min_pnt_dist* is the distance between the corresponding location of the sent point and the current location of the object.

As stated in Section 3.3.1, the *NEW_LABEL_REQ* message has information about the requester ID, side, and object coordinates in the requester coordinate system. Let $p_m^{(i)} = (x, y)$ denote the point sent by node i. When camera node j receives this message from node i, the corresponding location of $p_m^{(i)}$ in the view of C^j is calculated by using (1) as described in Section 2.1.1, and this corresponding location is denoted

by $p_{m,i}^{(j)}$. If the received *Side* information is not *middle*, the FOV line, $L_j^{i,s}$, corresponding to this side of the requester camera view is found. Then, the FG objects in the view of C^j, which were on the invisible side of $L_j^{i,s}$ and move in the direction of the visible side of the line, are determined. From those FG objects, the one that is closest to the point $p_{m,i}^{(j)}$ is found, and its label is sent back as the *Answer_label*. If, on the other hand, the received *Side* information is *middle*, then it means that this object appeared in the middle of the scene. In this case, FOV lines cannot be used, and the label of the object that is closest to the $p_{m,i}^{(j)}$ is sent back as the *Answer_label*. The *Min_pnt_dist* that is included in the reply message is the distance between $p_{m,i}^{(j)}$ and the location of the object that is closest $p_{m,i}^{(j)}$.

The proposed protocol also handles the case where the labels received from different cameras do not match. In this case, the label is chosen so that *Min_pnt_dist* is the smallest among all the reply messages.

3.3.4. Lost label reply case

If node j receives a message from node i, and the *Cmd_tag* of this message holds *LOST_LABEL_REQ*, then node j needs to send back a *lost_label* reply message to node i. The format of this message is

$$Cmd\_tag \quad Lost\_label \quad x\_reply \quad y\_reply. \tag{8}$$

Cmd_tag is a string that holds *LOST_LABEL_REP* indicating that this is a reply message to a *lost_label* request. *Lost_label* is an integer, which is the label of the tracker in C^i that could not be matched to its target object. When node j receives a *lost_label* request, it sends back the coordinates of the current location of the tracker with the label *Lost_label* as *x_reply* and *y_reply*. These coordinates are floats, and are in the coordinate system of C^j. When a reply message is received by node i, the corresponding point of the received location is calculated on the view of C^i as described in Section 2.1.1, and the location of the tracker is updated.

3.4. How to communicate

The steps so far provide an efficient protocol both by reducing the number of times a message must be sent as well as the message size. This part of the protocol addresses the issue of handling the communication and processing delays without using a centralized server. This process will henceforth be called the *system synchronization*. It should be noted that *system synchronization* is different from camera or input video synchronization as mentioned above.

The SCCS protocol utilizes *nonblocking* send and receive primitives for message communication. This effectively allows for a camera node to make its requests, noting the requests it made, and then continuing its processing with the expectation that the requestee will issue a reply message at some point later in execution. This is in contrast to *blocking* communication where the execution is blocked until a reply is received for a request. With *blocking* communication, the potential for parallel processing is reduced, as a camera node may be stuck waiting for its reply, while the processing program will likely require stochastic checks for messages. It is very difficult for each camera to predict when and how many messages will be received from other cameras. In the *nonblocking case*, checks for messages can take place in a deterministic fashion. Another possible problem with *blocking* communication is the increased potential for deadlocks. This can be seen by considering the situation where both cameras are making requests at or near simultaneous instances, as neither can process the other node's request while each waits for a reply.

System synchronization ensures the transfer of coherent vision data between cameras. To the best of our knowledge, existing systems do not discuss how to handle communication and processing delays without using blocking communications. Even if the cameras are synchronized or time-stamp information is available, communication and processing delays pose a problem for peer-to-peer camera systems. For instance, if camera C^i sends a message to camera C^j asking for information, it incurs a communication delay. When camera C^j receives this message, it could be on a frame behind camera C^i depending on the amount of processing its processor has to do, or it can be ahead of C^i due to the communication delay. As a result, the data received may not correspond to the data appropriate to the requesting camera's time frame. To alleviate this and achieve *system synchronization*, our protocol provides *synchronization points*, where all nodes are required to wait until every node has reached the same point. These points are determined based on a synchronization rate which will henceforth be called *synch_rate*. *Synchronization points* occur every *synch_rate* frames.

Between two synchronization points, each camera focuses on performing its local tracking tasks, saving the requests that it will make at the next synchronization point. When a new object appears in a camera view, a *new_label* request message is created for this object, and the object is assigned a *temporary* label. Since a camera node does not send the saved requests, and thus cannot receive a reply until the next synchronization point, the new object is tracked with this temporary label until receiving a reply back. Once a reply is received, the label of this object is updated.

Typical units of synchronization rate are time-stamp information for live camera input, or specific frame number for a recorded video. Henceforth, to be consistent, we refer to the number of video frames between each synchronization point when we use the terms synchronization rate or synchronization interval. There is no deterministic communication pattern for vision systems, so it is expected that the camera processors will frequently have to probe for incoming request messages. Although the penalty of probing is smaller than that of a send or receive operation, it is still necessary to decrease the number of probes because of power constraints. In order to decrease the amount of probing, we make each camera probe only when it finishes its local tasks and reaches a synchronization point.

Figure 8 shows a diagram of the system synchronization mechanism. This figure illustrates the camera states at the

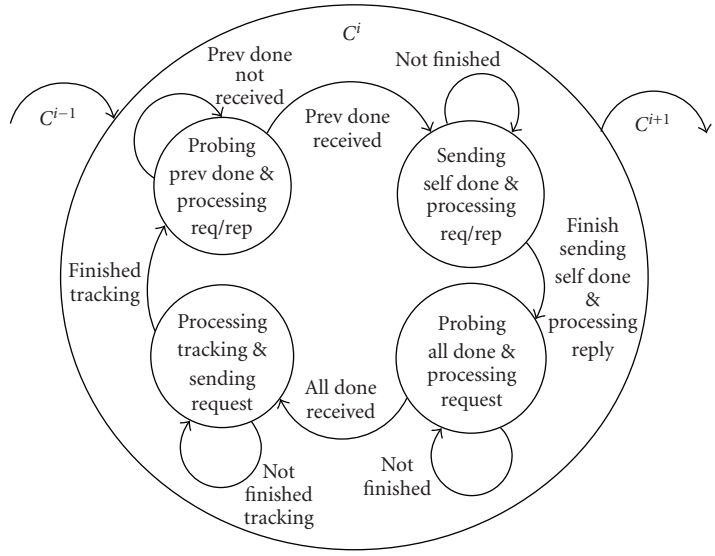


Figure 8: Camera states at the synchronization point.

synchronization point. In the first state, the camera finishes its local tracking, and the processor sends out all of its saved requests. Then, the camera enters the second state and begins to probe to see if a *done* message has been received from the previous camera. If not, this node probes for incoming requests from the other nodes and replies to them while waiting for the replies to its own requests. When the *done* message is received from the previous camera the camera enters the third state. When all of its own requests are fulfilled, it sends out a *done* message to the next camera. In the fourth state, each camera node still processes requests from other cameras, and keeps probing for the overall *done* message. Once it is received, a new cycle starts and the node returns back to the first state.

The *done* messages in our protocol are sent by using a *ring* type of message routing to reduce the number of messages. Thus, each node receives a *done* message only from its previous neighbor node and passes that message to the next adjacent node when it finishes its own local operations and has received replies to all its requests for that cycle. However, based on the protocol, all the cameras need to make sure that all the others already have finished their tasks before starting the next interval. Thus, a single pass of the *done* message is insufficient. If we have N cameras (C^i, $i = 0, \ldots, N-1$), a single pass of the *done* message will be from C^0 to C^1, C^1 to C^2, and so on. In this case, C^{i-1} will not know whether C^i has finished its task since it will only receive *done* messages from C^{i-2}. Thus, a second ring pass or a broadcast of an overall done message will be needed. In the current implementation, the overall done message is broadcasted from the first camera in the ring since the message is the same for every camera.

This protocol can handle the problems caused by communication delays and different processor loads and speeds, and incorporates variable synchronization capabilities, so as to allow flexibility with accuracy tradeoffs. As will be illustrated in Section 5 by Figures 18 and 19, the synchronization rate affects how soon a new label is received from the other cameras. In this protocol, the synchronization rate can be set by the end user depending on the system specification. Different synchronization rates are desirable in different system setups. For instance, for densely overlapped cameras, it is necessary to have a shorter synchronization interval because an object can be seen by several cameras at the same time, and each camera may need to communicate with others frequently. On the other hand, for loosely overlapped cameras, the synchronization interval can be longer since the probability for communication is lower and as a result, excess communication due to superfluous synchronization points is eliminated.

3.5. Comparison of the number of messages for a server-based scenario and for SCCS

3.5.1. A server-based system scenario for multicamera multiobject tracking

As stated before, server-based multicamera systems have a bandwidth scaling problem, and are limited by the server capacity. In order to illustrate the excessive number of messages and the load a server needs to handle, and compare these to the number of messages for our peer-to-peer communication protocol, we will introduce a *server-based system scenario* in this section. In this server-based system, the nodes keep the server updated by sending it messages for each tracker in their FOV. To make a fair comparison between this scenario and our communication protocol, we assume that these messages are sent at the synchronization points, which were defined in Section 3.4. In practice, due to different processing rates of the distinct processors coupled with communication delays, a server keeps the received data buffered to provide consistent data transfer between the nodes. However, this is not a practical approach since the

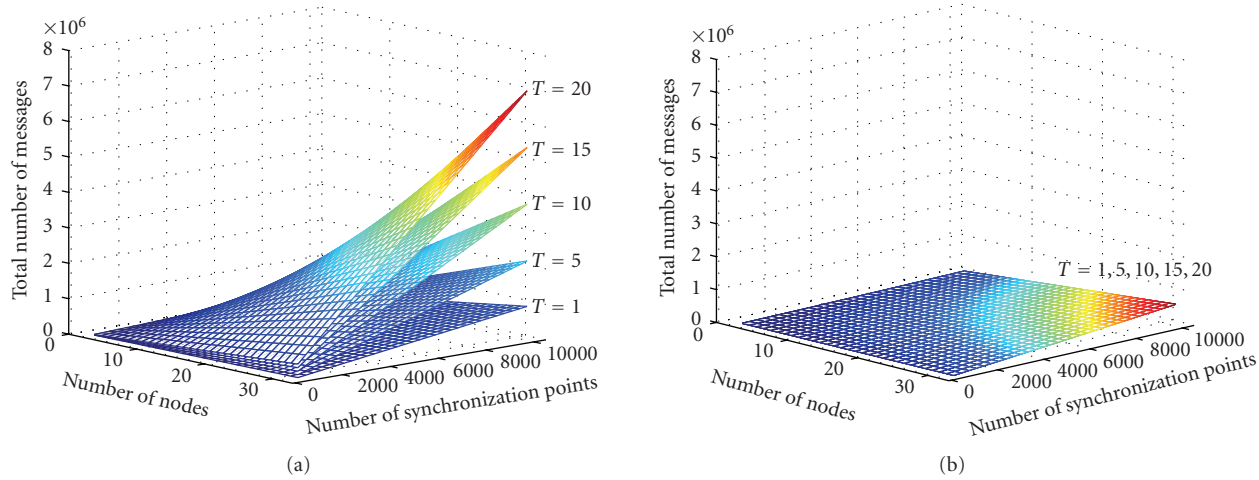


FIGURE 9: Comparison of the total number of messages for the server-based scenario and our peer-to-peer protocol: (a) and (b) show the total number of messages that need to be sent, for $T = 1, 5, 10, 15, 20$, with the server-based scenario and our peer-to-peer protocol, respectively. T is the number of trackers in a camera view. For our peer-to-peer protocol, the same surface represents all different values of T.

buffer size may need to be very large. Thus, we designed this scenario so that the server does not need a buffer. The nodes are required to wait at each synchronization point until they receive an overall done message from the server. At a synchronization point, each node needs to send a message for each tracker. These messages also indicate if the node has a request from any of the other nodes or not. Then, the server handles all these messages, determines the replies for each request, if there were any, and sends the replies to the corresponding nodes. The nodes update their trackers after receiving the replies, and acknowledge the server that they are done. After receiving a done message from all the nodes, the server sends an overall done message to the nodes so that nodes can move on. Based on this scenario, the total number of messages that are sent can be determined by using

$$M_{\text{server}} = S \times 2 \times N + \sum_{i=1}^{N} E_i + S \times \sum_{i=1}^{N} T_i, \tag{9}$$

where S is the number of synchronization points, N is the number of nodes/cameras and E_i is the total number of events that will trigger requests in the view of camera C^i. T_i is the total number of trackers in the view of C^i, and in this formula, without loss of generality, it is assumed that, for camera C^i, T_i remains the same during the video. In the server-based case, all these messages go through the server, and this argument will be revisited below.

On the other hand, for our protocol employed in SCCS, the total number of messages that are sent around is equal to

$$M_{\text{SCCS}} = S \times (2 \times N - 1) + 2 \times (N - 1) \times \sum_{i=1}^{N} E_i. \tag{10}$$

It should be noted that, when calculating M_{SCCS}, this equation considers the *worst* possible scenario, where it is assumed that all the cameras in the system view a common portion of the scene, and all the events happen in this overlapping region. This setup is highly unlikely since N cameras will be setup so that they are spatially distributed, and can cover a larger portion of the scene. In this *worst-case* scenario, in our peer-to-peer protocol, N nodes will send $N - 1$ request messages to the other nodes for E_i events and will receive $N - 1$ replies, hence the $2 \times (N - 1) \times \sum_{i=1}^{N} E_i$ term. At each synchronization point, each node will send a *done* message to its next neighbor in the ring, and the first node will send an overall done message to $N-1$ nodes, hence the $S \times (2 \times N - 1)$ term.

As seen from (10), contrary to the server-based scenario, the total number of messages sent around by our system is independent of the number of trackers in each camera view, since the communication is done in a peer-to-peer manner. This fact can also be seen by comparing Figures 9(a) and 9(b). These figures were obtained by setting $E_i = 20$ and $T_i = T$, for all i, where $T \in \{1, 5, 10, 15, 20\}$. It should be noted that Figures 9(a) and 9(b) are plotted so that their vertical axes have the same scale. As can be seen in Figure 9(b), for our peer-to-peer protocol, the same surface represents all the different values of T since the total number of messages is independent of the number of trackers in each camera view. In addition, (9) and (10), and Figure 9 show that the server-based scenario does not scale well with the increasing number of trackers in each camera view.

Now, we will compare the server-based system scenario with our peer-to-peer system in terms of the message *loads* on the individual nodes, which is a very important point. By load, we mean the number of messages that go through a node. In other words, we will compare the number of messages handled by the server in the server-based scenario and by the individual nodes of the SCCS. For the server-based scenario, all of the messages in (9) go through the server. Whereas, in SCCS, one ordinary node i has to send only $S + (N - 2) \times E_i + \sum_{k=1}^{N} E_k$ messages and receive

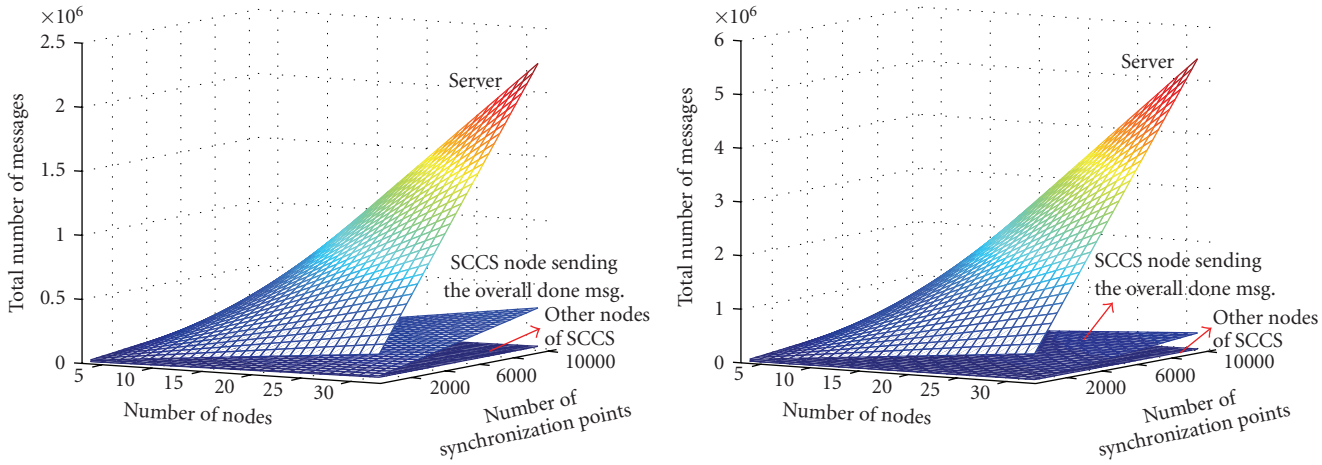


FIGURE 10: Comparison of the message loads: number of messages handled by the server in the server-based scenario and by the nodes of SCCS. (a) and (b) show the number of messages that are sent and received by the server and by the nodes of SCCS for $T = 5$ and $T = 15$, respectively.

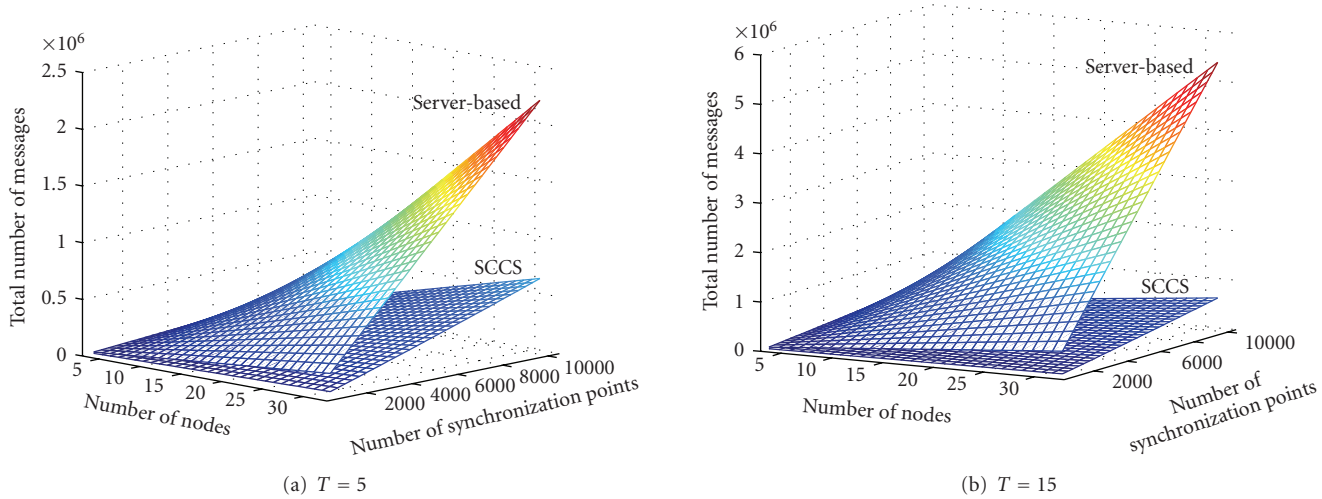
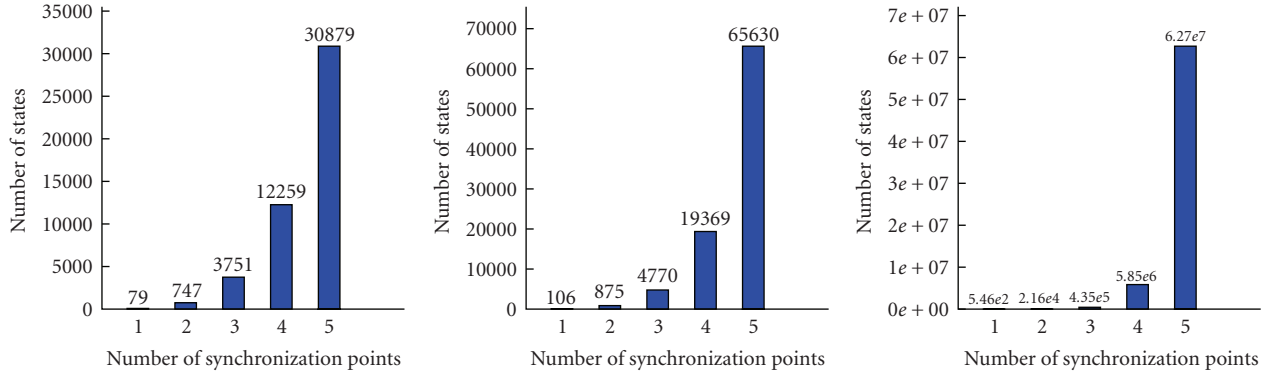


FIGURE 11: Comparison of the total number of messages for the server-based scenario and our peer-to-peer protocol for $T = 15$ and $T = 20$. For SCCS, it was assumed that all cameras have overlapping views and all events happen in overlapping region, which is highly unlikely.

$2\times S+(N-2)\times E_i+\sum_{k=1}^{N} E_k$ messages. The node j sending the overall done message has to send $S\times N+(N-2)\times E_j+\sum_{k=1}^{N} E_k$ messages, and receive $S+(N-2)\times E_j+\sum_{k=1}^{N} E_k$ messages. These numbers are plotted in Figures 10(a) and 10(b) for $T = 5$ and $T = 15$, respectively. As can be seen, the number of messages sent or received by the server is much larger than the number of messages sent or received by any node of the SCCS.

Figures 11(a) and 11(b) compare the *total* number of messages sent in the server-based scenario and our peer-to-peer protocol for $T = 5$ and $T = 15$, respectively, where T_i in (9) is set to be T, for all $i \in 1,\ldots,N$. It should be noted that this is the *total* number of messages. As stated before, in the server-based scenario, all of these messages go through the server. However, as seen in Figure 10, in our peer-to-peer protocol, each node handles a portion of this total number of messages. Thus, the load on a server is much larger than the load on the individual nodes of the SCCS. Another very important point to note is that the total number of messages for the SCCS is obtained for the worst possible scenario, where it is assumed that all the cameras in the system view a common portion of the scene, and all the events happen in this overlapping region. This is a highly unlikely case, since in real-life settings not all the cameras will have overlapping fields of view. Thus, in an N-node system, when an event happens in the view of camera C^i, it will only send request messages to the cameras which have overlapping fields of view with C^i, and can see this event. In other words, it will not need to send $N-1$ messages, and the $2\times(N-1)\times\sum_{i=1}^{N} E_i$ term in (10) will be much smaller.



(a) Scenario (a) (b) Scenario (b) (c) Scenario (c)

Figure 12: Number of states reached during verification of different communication scenarios.

For example, let us consider the case where there are $N = 32$ nodes in the system, $E_i = 40$, and every camera overlaps with 3 other cameras (not with $N-1 = 31$ cameras as in the worst-case scenario). Then there will be a 10% decrease in the total number of messages sent by SCCS compared to the worst-case calculations. Even with the worst case assumptions for the SCCS, the total number of messages sent is still much less than that of a server-based system, as can be seen in Figure 11.

Table 1: Comparison of exhaustive verification outputs for *synch_rate* of 1.

	No. of states	State-vector size (bytes)	Total memory usage (MB)	Depth
2 Proc. (a)	12259	496	3.017	159
3 Proc. (b)	19369	1252	3.217	182
3 Proc. (c)	5846880	1252	146.417	202

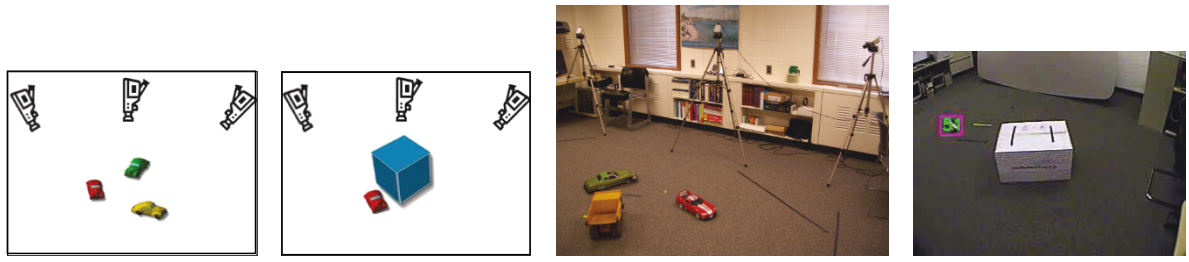
4. VERIFICATION OF THE PROTOCOL

Communicating between nodes in a peer-to-peer fashion and eliminating the use of a centralized server remove the single point of failure, decreases the number of messages sent around, and provide scalability and latency advantages. However, this requires a sophisticated communication protocol, which finds use in real-time systems having stringent requirements for proper system functionality. Hence, the protocol design for these systems necessitates transcending typical qualitative analysis using simulation; and instead, requires verification. The protocol must be checked to ensure that it does not cause unacceptable issues such as deadlocks and process starvation, and has correctness properties such as the system eventually reaching specified operating states. Formal verification methods of protocols can be derived from treating the individual nodes of a system as finite state automata. These then emulate communication through the abstraction of a channel.

SPIN is a powerful software tool used for the formal verification of distributed software systems [44]. It can analyze the logical consistency of concurrent systems, specifically of data communication protocols. A system is described in a modeling language called *Promela* (process meta language). Communication via message channels can be defined to be synchronous or asynchronous. Given a *Promela* model, *SPIN* can either perform random simulations of the system execution or it can perform *exhaustive* verification of correctness properties. It goes through all possible system states, enabling designers to discover potential flaws while developing protocols. This tool was used to analyze and verify the communication protocol used in SCCS and described in Section 3.

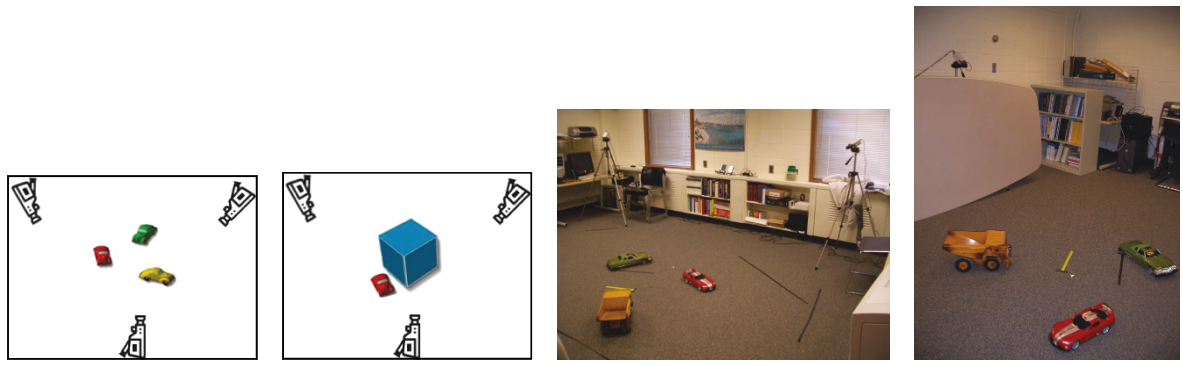
To analyze and verify the communication protocol of the SCCS, we first described our system by using *Promela*. We modeled three different scenarios: (a) a 2-processor system with full communication, where full communication means every processor in the system can send requests and replies to each other, (b) a 3-processor system, where the first processor can communicate with the second and third, the second processor can only communicate with the first, and the third one only replies to incoming requests, and (c) a 3-processor system with full communication. The reason of modeling scenario (b) is clarified below.

After modeling different scenarios, we first performed random simulations. With random simulation, every run may produce a different type of execution. In all the simulations of all three scenarios, all the processors of the model are terminated properly. However, each random simulation goes through one possible set of states. Thus, an exhaustive search of the state space is needed to guarantee that the protocol is error-free. We performed exhaustive verification of the three different scenarios with different synchronization rates. We also inserted an assertion into the model to ensure that a processor starts a new synchronization interval *only if* every processor in the system has sent a *done* message at the synchronization point. All of our three scenarios have been verified *exhaustively* with no errors. Table 1 shows the results obtained, where the *synch_rate* is 1, and there are 4 synchronization points. (a), (b), and (c) correspond to the scenarios described above. As can be seen in the table, when three processors are used with full communication, the number of states becomes very high



(a) (b) (c) (d)

Figure 13: (a) Locations of the cameras for the first camera setup; (b) environment state for the *lost_label* experiments; (c) a photograph of the first camera setup; (d) the view of the environment state from one of the cameras in the first setup.



(a) (b) (c) (d)

Figure 14: (a) Locations of the cameras for the second camera setup; (b) environment state for the *lost_label* experiments; (c) a photograph of the second camera setup showing two cameras placed on one side; (d) another photograph of the second camera setup showing the one camera placed on the opposite side.

compared to other scenarios, thus the search requires more memory. Scenario (b) was modeled so that we can compare scenario (c) to (b), and see the increase in the number of states and memory requirement. The total memory usage in the table is the "total actual memory usage" output of the *SPIN* verification. This is the amount after the compression performed by *SPIN*, and includes the memory used for a hash table of states.

Figure 12 shows the number of states reached with the three scenarios, and with different number of synchronization points. For the 3-processor and full communication scenario, the number of states increases very fast with increasing number of communication points. Since the memory requirement increases with the number of states, the scenario (c) requires the largest amount of memory for verification. In addition, when the *synch_rate* is increased, the number of states increases for the same number of synchronization points, as the requests of the local trackers are saved until the next synchronization point, and then sent out. These results illustrate that, as is well known in the field of communications, verification of complicated protocols is not a straightforward task. Also, careful modeling of the large systems having many possible states is very important for exhaustive verification.

Another important issue is how to handle system failure and reconfiguration. In order to address this issue with the ring type of message routing, we will incorporate the following steps into our communication protocol. We consider two cases of system failure: nonfunctioning cameras and nonfunctioning processors. For the nonfunctioning camera case, the processor itself can detect this type of failure, and then issue a broadcast message to other cameras which have overlapping fields of view with the failing camera. By checking the field of view overlap first, we will not broadcast to all the cameras in the network and thus decrease the number of messages. Those cameras will receive this failure indication message at the synchronization point. They will then recheck their pending label requests before sending them, and reassign, if possible, any messages pertaining to the nonfunctioning camera node.

Processor failure is more challenging to detect internally. To achieve this, we propose to employ a *lifetime checking* mechanism. We will treat each *done* message as a *heartbeat* signal, and use a *time-out* criterion to detect the processor failure. If processor P_i, which corresponds to camera i, does not receive correspondence from P_{i-1} after $t_{\text{time-out}}$, it will assume that there is a problem, and seek to determine whether the problem is with P_{i-1} or any processor before it. So P_i will send a checking message to P_{i-2}. If P_{i-2} has already sent out its *done* message to P_{i-1}, it will send back a response to P_i, and this will indicate a failure in P_{i-1}. Then P_i will issue a broadcast message to every camera in the group and

update the group information. Cameras having overlapping fields of view with P_{i-1} will ignore P_{i-1}. This will implicitly form an updated communication ring as well. On the other hand, if P_i sends a checking message to P_{i-2}, and P_{i-2} replies that it is still waiting for a *done* message from P_{i-3}, then P_{i-1} is potentially not at fault, and thus P_i would simply keep waiting. In this way, a fault at P_{i-3} or further left in the ring will be handled only by its right neighbor, which reduces complexity as well as overhead due to fault checking.

Although this checking mechanism consumes time, the elapsed time will be bounded by a limit set by the user. That is, the checking and updating will be completely finished by $t_{\text{time-out}} + t_{\text{non-functioning}}$, where $t_{\text{non-functioning}}$ is the time for sending and receiving one message (since all the camera nodes do the same checking at the same time) plus the time for a broadcast.

5. EXPERIMENTAL RESULTS

5.1. Camera setups

We have implemented SCCS on Linux using PC platforms and Ethernet. The system consists of a parallel computing cluster with uniprocessor nodes, each with a 1.5 GHz CPU. This section describes the results of experiments on a 3-camera 3-CPU system. Different types of experiments with different camera setups and video sequences of varying difficulty have been performed by using SCCS and the proposed communication protocol.

Figures 13 and 14 show the two different camera setups and two types of environment states used for the indoor experiments. We formed different environment states by placing or removing occluding structures, for instance a large box in our case, into the environment. As shown in Figures 13(a) and 14(a), we placed three cameras in two different configurations in a room. In the first configuration, all cameras are on one side of the room, whereas in the second configuration, two cameras are placed on one side, and the third one is placed on the opposite side. Figures 13(a) versus 13(b), and 14(a) versus 14(b) illustrate the two different environment states, that is, scenes with or without an occluding box. As seen in Figures 13(c), 13(d), 14(c), and 14(d), three remotely controlled cars/trucks have been used to experiment with various occlusion, merging, and splitting cases. We also captured different video sequences by operating one, two, or three cars at a time.

First, processing times of a single processor system and a distributed multicamera system incorporating peer-to-peer communication were compared. Figure 15 shows the speedup attained using our system relative to a uniprocessor implementation for two cases: processing input from two cameras and from three cameras. In the figure, processing times are normalized with respect to the uniprocessor case processing inputs from three cameras, which takes the longest processing time. As can be seen, the uniprocessor approach does not scale very well as processing the input from three cameras takes 3.57 times as long compared to processing input from two cameras. Whereas, in our case, processing inputs from three cameras by using three CPUs takes only 1.18 times as long compared to processing inputs from two cameras by using two CPUs. Hence, it is demonstrated that the execution time required is maintained, without significant increase, while adding the beneficial functionality of an additional camera. In addition, our approach provides 3.37× and 10.2× speedups for processing inputs from two and three cameras, respectively, compared to a uniprocessor system.

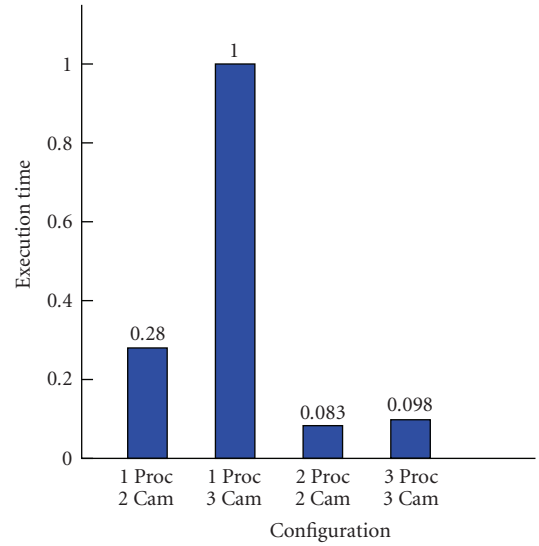


Figure 15: Comparison of the processing times required for processing inputs from two and three cameras by a uniprocessor system and by SCCS. 2Proc-2Cam and 3Proc-3Cam denote the times required by SCCS.

5.2. Waiting time experiments

In this set of experiments, we measured the average elapsed time between the instance an event occurs and the next synchronization point, where the reply of the request corresponding to this event is received. Henceforth, this elapsed time will be referred to as *waiting time*. For instance, if the *synch_rate* is 10, then the synchronization points will be located at frames $1, 11, 21, \ldots, 281, 291, 301 \ldots$, and so on. If a new object appears in a camera's FOV at frame 282, then the *waiting time* will be 9 frames, as the next synchronization point will be at frame 291.

Figure 16 shows the average *waiting time* for experiments performed with different video sequences with different *synch_rate* values. As can be seen, even when the *synch_rate* is 60 frames, the average *waiting times* are 33.06 and 26.3 frames for different video sequences.

5.3. Accuracy of the data transfer

In this set of experiments, we measured the accuracy of the data transfer and data updates. This accuracy is determined by the following formula:

$$\text{data\_transfer\_accuracy} = \frac{\#\text{correct\_updates}}{\text{total\_requests}} * 100, \qquad (11)$$

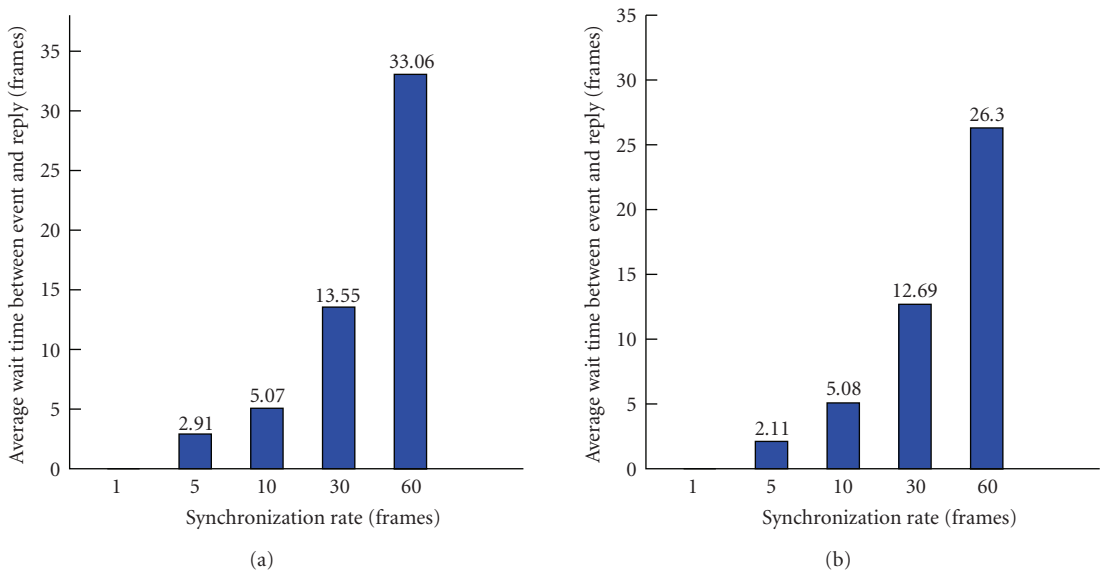


Figure 16: Waiting times for different videos and environment setups; (a) and (b) show the waiting times for the videos captured with indoor setup 1 and indoor setup 2, respectively.

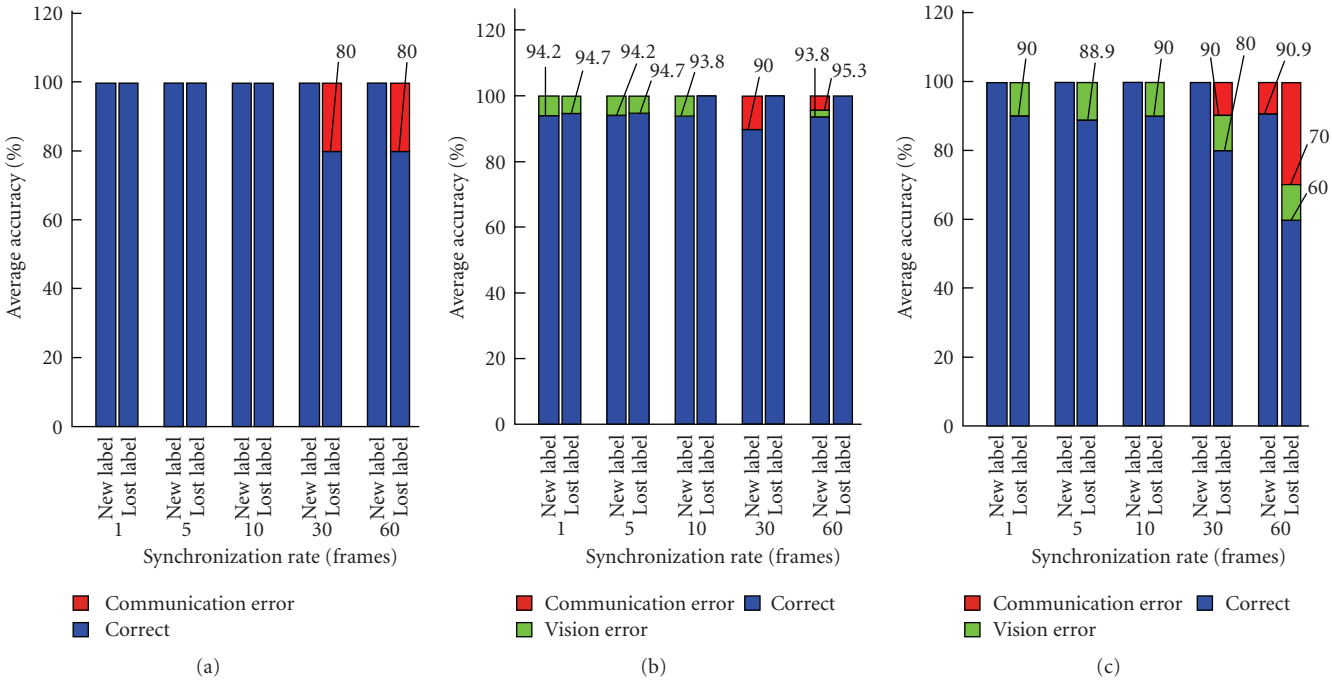
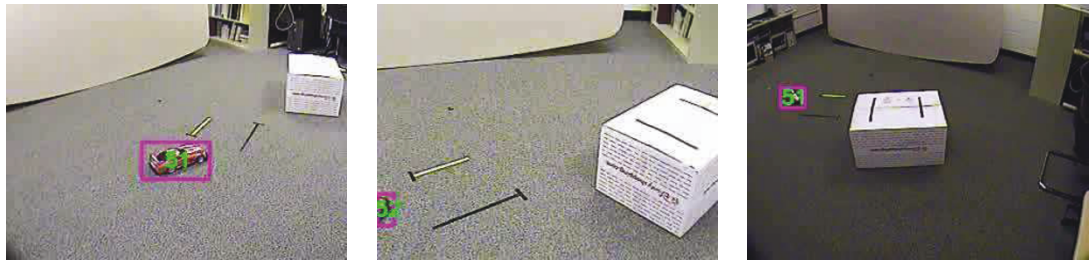


Figure 17: Average accuracy of the data transfer for indoor (a), (b) and outdoor (c) sequences.

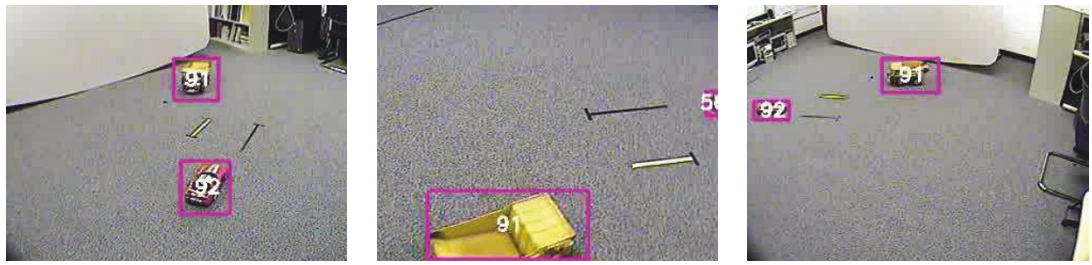
where #correct_updates represents the number of times a *new_label* or *lost_label* request is correctly fulfilled and the corresponding tracker is correctly updated (its label or its location). The determined accuracy values are shown in Figure 17. Red and green segments correspond to the error percentages caused by communication and vision algorithms, respectively. Communication errors depend on the *synch_rate*, because when *synch_rate* increases, nodes exchange data less often. Let the *synch_rate* be 60 frames. If node A loses an object at frame 121, for instance, it will be able to send a *LOST_LABEL_REQ* to node B at frame 180. During these 59 frames, the object of interest can disappear or merge with other objects in camera B's view, which will cause an error in the reply. This effect can be seen from Figure 17, where errors related to communication increase, in general, with increasing *synch_rate*. Overall, for a *synch_rate* of 1, the system achieves a minimum of 94.2% accuracy for the *new_label* requests/updates on both



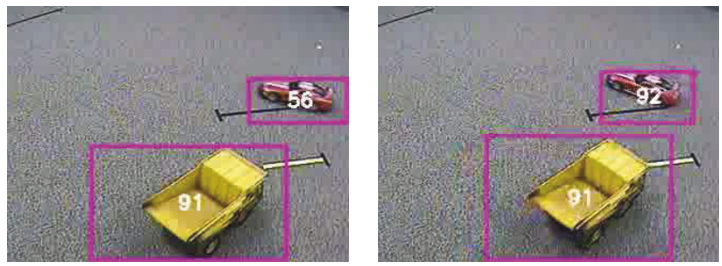
(a1) Camera 1—frame 2373 (b1) Camera 2—frame 2373 (c1) Camera 3—frame 2373

(b2) Camera 2—frame 2378 (b3) Camera 2—frame 2381

Figure 18: New label example for the first camera setup with a *synch_rate* of 10.



(a1) Camera 0—frame 1468 (b1) Camera 1—frame 1468 (c1) Camera 2—frame 1468

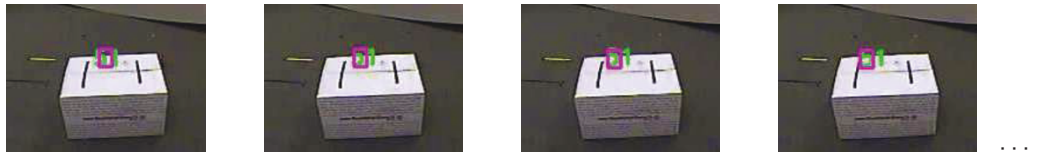


(b2) Camera 1—frame 1496 (b3) Camera 1—frame 1501

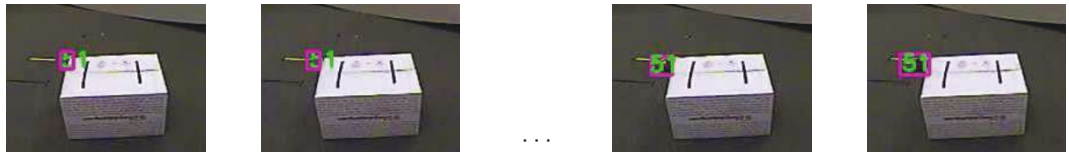
Figure 19: New label example for the second camera setup with a *synch_rate* of 60.

indoor and outdoor videos. For the *lost_label* requests, a minimum of 90% accuracy is achieved for both indoor and outdoor videos with a *synch_rate* of 1. Further, even with allowing the processors to operate up to 2 seconds without communication, a minimum of 90% accuracy is still attained for *new_label* requests with indoor sequences, while 90.9% accuracy is obtained for the outdoor sequence. Again, with allowing the processors to operate up to 2 seconds without communication, a level of 80% or higher accuracy is attained for *lost_label* requests with indoor sequences, while 60% accuracy is obtained for the outdoor sequence.

Figures 18, 19, and 23 illustrate the success of the consistent labeling algorithm, where the same objects are given the consistent labels in different camera views. Figures 18 and 19 show examples of receiving the label of a new tracker from the other nodes, and updating the label of the tracker in the current view accordingly. For Figure 18, the *synch_rate* is 10. As can be seen in Figure 18(b1), when



(a1) Frame 681 (a2) Frame 686 (a3) Frame 691 (a4) Frame 696 . . .



(a5) Frame 716 (a6) Frame 721 . . . (a7) Frame 736 (a8) Frame 741

FIGURE 20: Lost object example for the first camera setup with a *synch_rate* of 5.

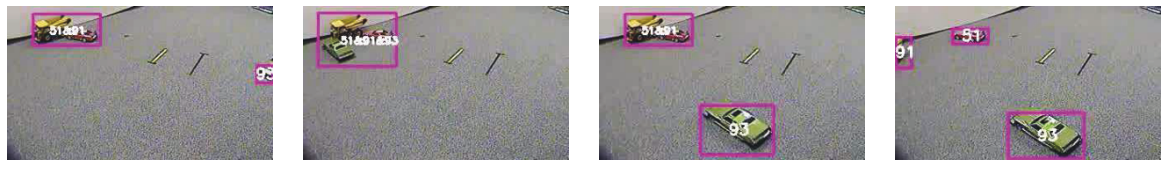
(a1) Camera 1—frame 810 (b1) Camera 2—frame 810 (c1) Camera 3—frame 810

(a1) Frame 810 (a2) Frame 832 (a3) Frame 852 (a4) Frame 914

FIGURE 21: Lost object example for the second camera setup with a *synch_rate* of 1.

(a) Frame 1876 (b) Frame 2001 (c) Frame 2013 (d) Frame 2045

FIGURE 22: Lost object example for the outdoor sequence with a *synch_rate* of 1.



(a1) Cam. 1—frame 1902 (a2) Cam. 1—frame 2151 (a3) Cam. 1—frame 2301 (a4) Cam. 1—frame 2476

(b1) Cam. 3—frame 1902 (b2) Cam. 3—frame 2151 (b3) Cam. 3—frame 2301 (b4) Cam. 3—frame 2476

FIGURE 23: Successfully resolving the merging/splitting of three objects.

(a) Frame 411 (b) Frame 431 (c) Frame 436

Figure 24: Successfully resolving a merge/split event on a PETS sequence.

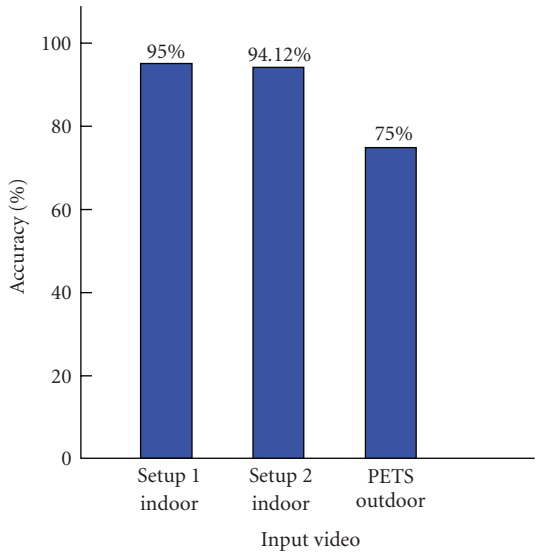


Figure 25: Accuracy of handling merge/split cases for indoor and outdoor sequences.

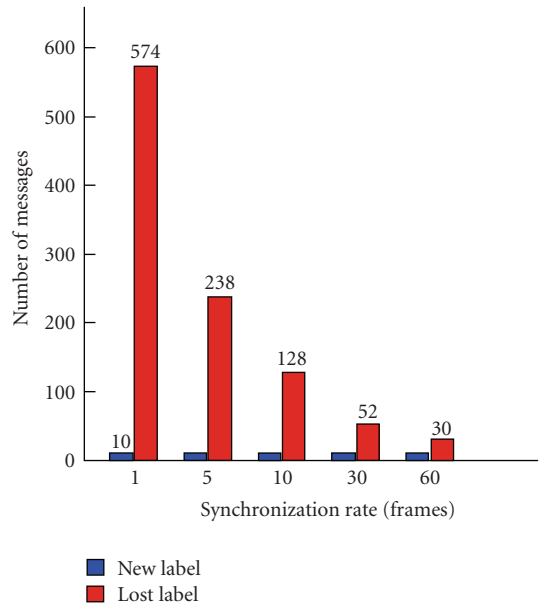


Figure 26: Number of *new_label* and *lost_label* requests for different *synch_rates*.

the car first appears in the view of camera 2, it is given a *temporary* label of 52, and is tracked with this label until the next synchronization point. Then, the correct label is received from the other nodes in the system and the label of the tracker in the view of camera 2 is updated to be 51 as seen in Figure 18(b3). Figure 19 is another example for a *synch_rate* of 60 for the second camera setup. Again, the label of the tracker, created at frame 1468 and given a temporary label of 56, is updated successfully at frame 1501 from the other nodes in the system.

Figures 20, 21, and 22 show examples of updating the location of a tracker, whose target object is lost, from the other nodes. For Figure 20, the *synch_rate* is 5, and the views of the three cameras are as seen in Figures 18(a1), 18(b1) and 18(c1). As seen in Figures 20(a1) through 20(a8), the location of the car behind the box is updated every 5 frames from the other nodes, until it reappears. Figure 21 is another example for a *synch_rate* of 1 for the second camera setup. The location of the tracker is updated at every frame. Figure 22 shows an example, where the location of people occluded in an outdoor sequence is updated.

Figures 23 and 24 show examples of SCCS dealing with the merge/split cases on a single camera view for indoor and outdoor videos, respectively. The accuracy of giving the correct labels to objects after they split is displayed in Figure 25.

Figure 26 shows the number of *new_label* and *lost_label* requests for different synchronization rates for the video captured by the first camera setup with the box placed in the environment. As expected, with a *synch_rate* of 1, a *lost_label* request is sent at each frame as long as the car is occluded behind the box. Thus, the number of *lost_label* requests is the highest for the *synch_rate* of 1, and decreases with increasing *synch_rate*.

6. CONCLUSIONS

This paper has presented the scalable clustered camera system, which is a *peer-to-peer* multicamera system for multiple object tracking. Each camera is connected to a CPU, and individual nodes communicate with each other directly eliminating the need for a centralized server. Instead of transferring control of tracking jobs from one camera to another, each camera in the presented system keeps its

own trajectories for each target object, which provides fault tolerance. A fast and robust tracking algorithm was proposed to perform tracking on each camera view, while maintaining consistent labeling.

Peer-to-peer systems require sophisticated communication protocols that can handle communication and processing delays. These protocols need to be evaluated and verified against potential deadlocks, and their correctness properties need to be checked. We introduced a novel communication protocol designed for peer-to-peer vision systems, which can handle the communication and processing delays. The reasons of processing delays include heterogenous processors, different loads at different processors, and instruction and task scheduling within the node processing unit. The protocol presented in this paper incorporates variable synchronization capabilities. Moreover, compared to server-based systems, it decreases the number of messages that a single node has to handle as well as the total number of messages that need to be sent considerably. We then analyzed and exhaustively verified this protocol, without any errors or redundancies, by using the *SPIN* verification tool.

Video sequences with varying levels of difficulty have been captured by using different camera setups and environment states. Different experiments were performed to obtain the speed up provided by SCCS, to measure average data transfer accuracy and average *waiting time*. Experimental results demonstrate the success of the SCCS, with high data transfer accuracy rates.

REFERENCES

[1] N. Atsushi, K. Hirokazu, H. Shinsaku, and I. Seiji, "Tracking multiple people using distributed vision systems," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '02)*, vol. 3, pp. 2974–2981, Washington, DC, USA, May 2002.

[2] D. Beymer, P. McLauchlan, B. Coifman, and J. Malik, "A real-time computer vision system for measuring traffic parameters," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '97)*, pp. 495–501, San Juan, Puerto Rico, USA, June 1997.

[3] M. Bramberger, A. Doblander, A. Maier, B. Rinner, and H. Schwabach, "Distributed embedded smart cameras for surveillance applications," *Computer*, vol. 39, no. 2, pp. 68–75, 2006.

[4] Q. Cai and J. K. Aggarwal, "Automatic tracking of human motion in indoor scenes across multiple synchronized video streams," in *Proceedings of the 6th IEEE International Conference on Computer Vision (ICCV '98)*, pp. 356–362, Bombay, India, January 1998.

[5] Q. Cai and J. K. Aggarwal, "Tracking human motion in structured environments using a distributed-camera system," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 11, pp. 1241–1247, 1999.

[6] S. Calderara, R. Cucchiara, and A. Prati, "Group detection at camera handoff for collecting people appearance in multi-camera systems," in *Proceedings of the IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS '06)*, pp. 36–41, Sydney, Australia, November 2006.

[7] S. Calderara, R. Cucchiara, and A. Prati, "Bayesian-competitive consistent labeling for people surveillance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 354–360, 2008.

[8] T.-H. Chang and S. Gong, "Tracking multiple people with a multi-camera system," in *Proceedings of the IEEE Workshop on Multi-Object Tracking (WOMOT '01)*, pp. 19–26, Vancouver, Canada, July 2001.

[9] R. T. Collins, A. J. Lipton, T. Kanade, et al., "A system for video surveillance and monitoring: VSAM final report," Tech. Rep. CMU-RI-TR-00-12, Robotics Institute, Carnegie Mellon University, Pittsburgh, Pa, USA, May 2000.

[10] R. T. Collins, A. J. Lipton, H. Fujiyoshi, and T. Kanade, "Algorithms for cooperative multisensor surveillance," *Proceedings of the IEEE*, vol. 89, no. 10, pp. 1456–1477, 2001.

[11] T. Ellis, "Multi-camera video surveillance," in *Proceedings of the 36th IEEE Annual International Carnahan Conference on Security Technology (ICCST '02)*, pp. 228–233, Atlantic City, NJ, USA, October 2002.

[12] W. Hu, M. Hu, X. Zhou, T. Tan, J. Lou, and S. Maybank, "Principal axis-based correspondence between multiple cameras for people tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 663–671, 2006.

[13] O. Javed, S. Khan, Z. Rasheed, and M. Shah, "Camera handoff: tracking in multiple uncalibrated stationary cameras," in *Proceedings of the Workshop on Human Motion (HUMO '00)*, pp. 113–118, Austin, Tex, USA, December 2000.

[14] P. H. Kelly, A. Katkere, D. Y. Kuramura, S. Moezzi, S. Chatterjee, and R. Jain, "An architecture for multiple perspective interactive video," in *Proceedings of the 3rd ACM International Conference on Multimedia (MULTIMEDIA '95)*, pp. 201–212, San Francisco, Calif, USA, November 1995.

[15] V. Kettnaker and R. Zabih, "Bayesian multi-camera surveillance," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '99)*, vol. 2, pp. 253–259, Fort Collins, Colo, USA, June 1999.

[16] S. Khan and M. Shah, "Consistent labeling of tracked objects in multiple cameras with overlapping fields of view," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 10, pp. 1355–1360, 2003.

[17] J. Krumm, S. Harris, B. Meyers, B. Brumitt, M. Hale, and S. Shafer, "Multi-camera multi-person tracking for EasyLiving," in *Proceedings of the 3rd IEEE International Workshop on Visual Surveillance (VS '00)*, pp. 3–10, Dublin, Ireland, July 2000.

[18] L. Lee, R. Romano, and G. Stein, "Monitoring activities from multiple video streams: establishing a common coordinate frame," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 758–767, 2000.

[19] C. Madden, E. D. Cheng, and M. Piccardi, "Tracking people across disjoint camera views by an illumination-tolerant appearance representation," *Machine Vision and Applications*, vol. 18, no. 3-4, pp. 233–247, 2007.

[20] K. Nguyen, G. Yeung, S. Ghiasi, and M. Sarrafzadeh, "A general framework for tracking objects in a multi-camera environment," in *Proceedings of the 3rd International Workshop on Digital and Computational Video (DCV '02)*, pp. 200–204, Cleanvater Beach, Fla, USA, November 2002.

[21] N. T. Nguyen, H. H. Bui, S. Venkatesh, and G. West, "Recognizing and monitoring high-level behaviors in complex spatial environments," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '03)*, vol. 2, pp. 620–625, Madison, Wis, USA, June 2003.

[22] N. T. Nguyen, S. Venkatesh, G. West, and H. H. Bui, "Multiple camera coordination in a surveillance system," *Acta Automatica Sinica*, vol. 29, no. 3, pp. 408–422, 2003.

[23] H. Pasula, S. J. Russell, M. Ostland, and Y. Ritov, "Tracking many objects with many sensors," in *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI '99)*, pp. 1160–1171, Stockholm, Sweden, July-August 1999.

[24] I. Pavlidis, V. Morellas, P. Tsiamyrtzis, and S. Harp, "Urban surveillance systems: from the laboratory to the commercial world," *Proceedings of the IEEE*, vol. 89, no. 10, pp. 1478–1497, 2001.

[25] A. Utsumi, H. Mori, J. Ohya, and M. Yachida, "Multiple-camera-based human tracking using non-synchronous observations," in *Proceedings of 4th Asian Conference on Computer Vision (ACCV '00)*, pp. 1034–1039, Taipei, Taiwan, January 2000.

[26] S. Velipasalar, L. M. Brown, and A. Hampapur, "Specifying, interpreting and detecting high-level, spatio-temporal composite events in single and multi-camera systems," in *Proceedings of the International Workshop on Semantic Learning Applications in Multimedia (SLAM) in Conjunction with IEEE Conference on Computer Vision and Pattern Recognition*, pp. 110–117, New York, NY, USA, June 2006.

[27] S. Velipasalar and W. Wolf, "Recovering field of view lines by using projective invariants," in *Proceedings of the IEEE International Conference on Image Processing (ICIP '04)*, vol. 5, pp. 3069–3072, Singapore, October 2004.

[28] S. Velipasalar and W. Wolf, "Multiple object tracking and occlusion handling by information exchange between uncalibrated cameras," in *Proceedings of the IEEE International Conference on Image Processing (ICIP '05)*, vol. 2, pp. 418–421, Genova, Italy, September 2005.

[29] S. Velipasalar, J. Schlessman, G.-Y. Chen, W. Wolf, and J. P. Singh, "SCCS: a scalable clustered camera system for multiple object tracking communicating via message passing interface," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME '06)*, pp. 277–280, Toronto, Canada, July 2006.

[30] X. Yuan, Z. Sun, Y. Varol, and G. Bebis, "A distributed visual surveillance system," in *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance (AVSS '03)*, pp. 199–204, Miami, Fla, USA, July 2003.

[31] S. Funiak, M. Paskin, C. Guestrin, and R. Sukthankar, "Distributed localization of networked cameras," in *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN '06)*, pp. 34–42, Nashville, Tenn, USA, April 2006.

[32] B. P. L. Lo, J. Sun, and S. A. Velastin, "Fusing visual and audio information in a distributed Intelligent surveillance system for public transport systems," *Acta Automatica Sinica*, vol. 29, no. 3, pp. 393–407, 2003.

[33] J. A. Watlington and V. M. Bove Jr., "A system for parallel media processing," *Parallel Computing*, vol. 23, no. 12, pp. 1793–1809, 1997.

[34] C. Karlof, N. Sastry, and D. Wagner, "Cryptographic voting protocols: a systems perspective," in *Proceedings of the 14th USENIX Security Symposium (USENIX Security '05)*, pp. 33–49, Baltimore, Md, USA, August 2005.

[35] N. Evans and S. Schneider, "Analysing time dependent security properties in CSP using PVS," in *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS '00)*, pp. 222–237, Toulouse, France, October 2000.

[36] V. Vanackère, "The TRUST protocol analyser, automatic and efficient verification of cryptographic protocols," in *Proceedings of the Verification Workshop (VERIFY '02)*, pp. 17–27, Copenhagen, Denmark, July 2002.

[37] H. Bowman, G. Faconti, and M. Massink, "Specification and verification of media constraints using UPPAAL," in *Proceedings of the 5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems (DSV-IS '98)*, pp. 261–277, Abingdon, UK, June 1998.

[38] T. Sun, K. Yasumoto, M. Mori, and T. Higashino, "QoS functional testing for multi-media systems," in *Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '03)*, pp. 319–334, Berlin, Germany, September-October 2003.

[39] S. Velipasalar, C.-H. Lin, J. Schlessman, and W. Wolf, "Design and verification of communication protocols for peer-to-peer multimedia systems," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME '06)*, pp. 1421–1424, Toronto, Canada, July 2006.

[40] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, Cambridge, UK, 2001.

[41] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '99)*, vol. 2, pp. 246–252, Fort Collins, Colo, USA, June 1999.

[42] D. Comaniciu, V. Ramesh, and P. Meer, "Real-time tracking of non-rigid objects using mean shift," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '00)*, vol. 2, pp. 142–149, Hilton Head Island, SC, USA, June 2000.

[43] The MPI Standard, September 2001, http://www.unix.mcs.anl.gov/mpi.

[44] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, Boston, Mass, USA, 2004.