Mobile Virtual Sensors: A Scalable Programming and Execution Framework for Smart Surveillance (Position Paper)

Rajnish Kumar[†], Junsuk Shin[†], Liviu Iftode[‡], Umakishore Ramachandran[†]
[†] Georgia Institute of Technology
[‡] Rutgers University
rajnish@cc.gatech.edu, jshin@cc.gatech.edu, iftode@cs.rutgers.edu, rama@cc.gatech.edu

Abstract

Sensors of various modalities and capabilities have become ubiquitous in our environment. Technological advances and the low cost of such sensors enable the deployment of large-scale camera networks in large metropolis such as London and New York. Multimedia algorithms for analyzing and drawing inferences from video and audio have also matured tremendously in recent times. Despite all these advances, large-scale reliable systems for sensor-based applications such as surveillance are yet to become commonplace. Why is that?

Clearly, this is an area of great importance to the sensor network community. We argue that building effective smart surveillance systems requires a different approach to the problem, with a focus on programmability and scalability. While considerable progress has been made in computer vision algorithms, such advances cannot translate to deployment in the large until adequate system abstractions and resource management techniques are in place to ensure their performance.

We present the system requirements and design principles for smart surveillance systems, and propose a distributed systems architecture based on a novel abstraction called Mobile Virtual Sensor for scalable surveillance. Using this architecture as a springboard for further discussion, we explore various research challenges to be solved for realizing effective smart surveillance systems. Our preliminary experiments based on a camera sensor network prototype show considerable benefits in resource usage by using the proposed architecture.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

HotEmNets'08, June 2–3, 2008, Charlottesville, Virginia, USA
Copyright 2008 ACM ISBN 978-1-60558-209-2/08/0006...$5.00

General Terms

Design, Experimentation

Keywords

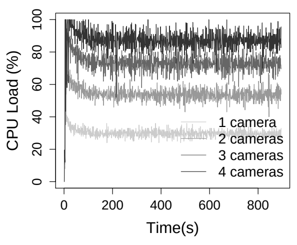
Mobile, Sensor Network, Systems, Surveillance, Cameras

1 Introduction

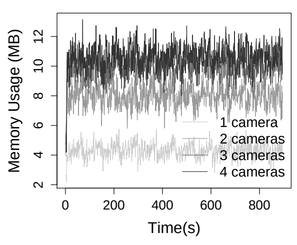
Surveillance technologies have dramatically evolved during the last decade. Although still the most common surveillance method in use, traditional video surveillance based on Closed Circuit Television (CCTV) requires continuous monitoring by humans, which is both expensive and error prone. As CCTV technology has advanced to include IP digital cameras, an opportunity to offload the monitoring task to computers has become possible and has paved the way towards semi and completely automatic object tracking systems. Advances in sensor technologies and networking have enriched the diversity and comprehensibility of the data that can be extracted from the observed scene. Clearly, surveillance has evolved from a manual, human-operated activity to a complex computing problem, with substantially difficult challenges and the potential to have a major impact on our daily lives.

Smart or *intelligent surveillance* are the terms used to describe the combination of sensing and computing for the purpose of automatic identification of interesting objects/targets and/or suspicious behaviors. Most smart video surveillance systems take a centralized approach in processing the camera streams, where cameras have little if any computing capabilities. On these systems, intelligent video processing software (commonly referred to as video analytics) uses image recognition algorithms to actively and rapidly scan through video feeds.

Providing high scalability in the presence of large amount of live streaming data and their real-time processing is still a huge challenge. The severity of infrastructure overload is apparent for camera systems because image dissemination and processing tasks are very resource intensive. Consider, for example, a simple surveillance system that does motion sensing and JPEG encoding/decoding. Figure 1 shows the processing requirements for such a system using a centralized set up: cameras produce data at 5 frames/second with a 320x240 resolution; image processing happens on a 1.4GHz Pentium processor. The results show that the above cen-



(a) CPU Load



(b) Memory Usage

Figure 1. Resource usage in a centralized camera network: cameras produce data at 5 fps with 320x240 resolution. Image processing and RFID reading happen on a 1.4GHz Pentium processor.

tralized setup cannot scale beyond four cameras (the CPU load is nearly 100%.) If we increase the video quality (frames/second and resolution), even a high-end computing resource will be unable to process more than a few cameras. Clearly, scaling up to a large number of cameras (on the order of 100's or 1000's) warrants a distributed architecture.

Programmability of smart surveillance systems to support the development of large-scale sensing applications is another significant challenge. Simple and intuitive programming and execution models are absolutely essential to allow the domain experts (e.g., specialists in video analytics) to focus on the algorithmic aspects of the application rather than the physical layout of cameras, configuration and resource management, fault tolerance, etc. Similar challenges exist in ensuring system reliability, extensibility, and security for smart surveillance.

To improve the programmability and scalability of surveillance systems, in this position paper, we propose a novel abstraction called *mobile virtual sensor* (MVS). MVS provides an interface through which programmers can develop surveillance and target tracking applications. As the name suggests, an MVS can be thought as a virtual sensor that follows physical objects, thus hiding all of the complexities involved in implementing resource management and tracking logic across distributed sensing nodes. The design of MVS abstraction is based on the basic principle of *selective attention*, i.e. at any given time, only a few sensors are interesting and actually need attention for tracking targets. By reducing the number of sensor streams that are to be processed, MVS approach to programming will reduce overall resource requirement. To further improve scalability, MVS employs feature prioritization and dynamic resource mapping strategies. In the rest of this paper, first we explore the programming model using MVS abstraction, and then we explain different aspects of the MVS interface design.

2 A Smart Surveillance Scenario

To set the stage for our proposed model, we describe a small part of a possible object identification and tracking application at an airport, which is purposely exaggerated for illustration purposes. Cameras are deployed at gates, walkways, and baggage claim areas. An automatic tracking system is configured to spawn "tracking tasks" whenever "unusual" behavior is detected at certain points of interest, such as doors, gate areas, etc. For each type of tracking task, a starting predicate is defined to trigger alerts at different intensity levels. We assume that an "orange" label defines a low-intensity alert, while a "red" one represents a high-intensity alert. For instance, an orange "gate alert" task is triggered when an unauthorized individual is detected within a certain proximity of the gate entrance (e.g., the individual does not carry an electronic badge).

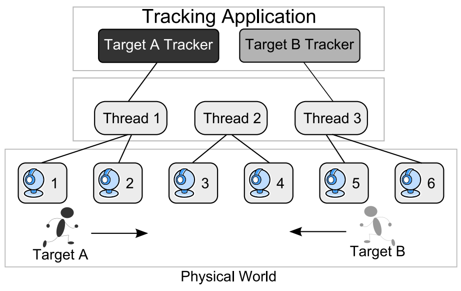
Once the tracking task is spawned, the individual is defined as a target and is followed throughout the airport until he/she leaves the airport either by boarding a plane or leaving through an airport exit. During this surveillance, he/she is continuously monitored by at least two cameras within his proximity and images of him/her are periodically shown to security personnel (either on their PDAs or on orange "gate alert" monitors in a control room). If the individual gets close to the entrance of a gate for a second time, a red "gate alert" is triggered and security personnel are notified. The security personnel could then monitor the behavior of the individual and decide whether to continue the tracking task, or to terminate it. Finally, if two targets at orange alerts meet, the alert level is escalated to red.

During normal airport activity, several orange gate alert tasks are always executing. At any time, a security agent could manually spawn a tracking task by selecting a target (either from his PDA or from one of the security monitors). Security personnel may also enter the suspect's description as a set of features, e.g. color, height, weight, or other suspect-specific attributes without indicating any location. The suspect description is fed as an input to the surveillance system for continuous real-time monitoring (and displaying to the requesting security personnel) of possible suspects meeting such description.

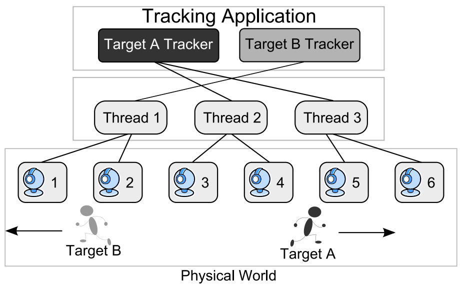
3 The Smart Surveillance Model

The computer vision community has proposed many object identification and tracking algorithms that could be directly applied to the scenario described above. A typical object tracking architecture consists of two main levels: image level for background subtraction and segmentation, and blob level for feature extraction and matching to generate a tracking digest of a specific target. The above two levels help to identify the computational logic needed for each of the camera streams. The output from the second level, namely, the tracking digests, from different camera streams is shared for further refinement and improved tracking accuracy by using the topological relationship among the installed cameras. The above architecture can easily be realized in the small (i.e., for a few 10's of camera streams) by implementing the computational logic to be executed on each of the camera streams, and the output to be centrally analyzed for correlation and refinement. Indeed, there are video analytic solution providers [2] that peddle mature commercial products for such scenarios.

Although there is no technological limitation preventing a network of cameras and sensors from executing the tracking scenario described above, programming such scenarios in the large is a hard problem, especially when the require-

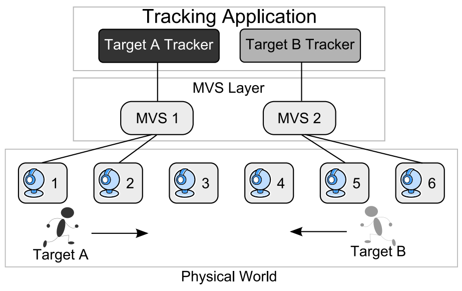


(a) Snapshot 1

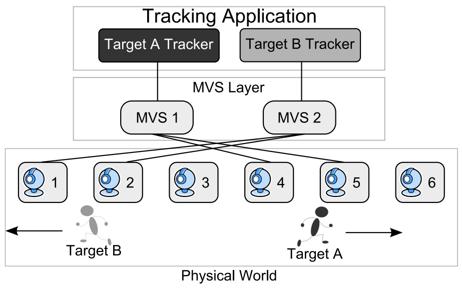


(b) Snapshot 2

Figure 2. Target tracking using traditional programming approach.



(a) Snapshot 1



(b) Snapshot 2

Figure 3. Target tracking using MVS programming model.

ments for critical systems, namely, scalability, reliability, extensibility and security, must be guaranteed. The gap between the image processing algorithms and their feasible deployment in a viable large-scale surveillance system is huge.

Programming the tracking application we described earlier is particularly challenging due to the lack of right abstractions. Although the "thread model" of programming serves the needs of parallel programming (especially for technical applications), we argue that this is a too low level abstraction for building large-scale smart surveillance systems. Besides, at the system level, a thread merely captures resource needs of a tracking task in terms of processor cycles. On the other hand, a tracking task requires several resources simultaneously: sensors (perhaps not just cameras if the task is working on multi-modal algorithms), network resources, memory, and processing cycles.

A simple-minded implementation of the above tracking application based on the thread model would assign a thread to a subset of the cameras and fuse the data streams to detect events of interest like in Figure 2. Such an approach would force the application to continuously map targets to threads of execution, making it difficult to perform resource allocation effectively. Moreover, an application programmer would have to deal with thread management, data synchronization, buffer handling, and exceptions (such as time-outs while waiting for data from sensor data for a fusion function) - all with the increased complexity of a large-scale loosely coupled distributed system. In dealing with these programming challenges, performance typically has to take a back seat. Clearly, threads are the wrong level of abstraction for this domain since they do not include the right set of semantics for dealing with the dynamic requirements of target tracking, which is central to smart surveillance.

3.1 Design Principles

We identified four main principles to be considered in the design of a scalable surveillance system:

1. Selective attention: At any given time, only a few cameras will actually need attention. However, identifying the camera streams of interest will depend on many factors, including the topological map of the area, projected trajectories of the targets, and other possible application-specific attributes.
2. State management across streams: In order to correlate tracking information across different camera streams, or to support multi-sensor fusion, state sharing across different sensors is very important. As the object of interest moves, the set of sensors to be correlated also changes. Providing communication service among such a dynamic set of nodes is another challenge for the application programmer.
3. Feature prioritization: In order to get an accurate interpretation of scenes, a large set of features must be extracted for object identification. However, considering the limitation in available resources, there exists a trade-off between performance and the feature set to be extracted.
4. Resource mapping: Once the system identifies the set of interesting camera streams, the stream processing logic needs to be scheduled on available computational resources. Thus, efficient resource mapping in such a dynamic environment is a challenge.

3.2 Mobile Virtual Sensors

Following the design principles listed above, we propose a novel architecture for smart surveillance systems based on Mobile Virtual Sensors (MVS). A Mobile Virtual Sensor is an abstract sensing unit that follows a target defined by the application. Applications define mobile virtual sensor classes for particular types of targets by specifying the starting, tracking, and termination *predicates*. When the conditions indicated in the starting predicates are met, an MVS is spawned. An MVS tracks the target it is associated with at all times. The resources needed for an MVS (sensor streams, processing, network, memory) are dynamically made available in the proximity of the mobile target.

Figures 2-3 show the difference between programming using the MVS abstraction and the conventional thread model described earlier. In the conventional approach, the threads do not understand targets, hence the application has to control allocation of threads, cameras and other resources to targets (which is what it ultimately cares about). On the other hand, the MVS model dramatically simplifies tracking application programming by associating the unit of execution (MVS) with a (possibly moving) target. This is achieved by adding a layer of indirection between the application and the sensors, which performs dynamic and transparent mapping of the MVS to physical sensors according to certain conditions/predicates that define the target (see Figure 3). In this way, the MVS-based programming separates the application concern of performing image analysis from the two system concerns, namely, discovering the physical sensors that can deliver the necessary video stream and managing the surveillance system resources efficiently.

The MVS architecture consists of three main components: the MVS module for selective attention and state management across streams, a prioritization module for feature prioritization, and a placement module for resource mapping. The prioritization and placement modules are designed using heuristics described in prior work [8, 7]. This paper focuses on the design of the MVS layer. In order to support selective attention, topological map along with the location of sensors and the possible locations of targets are used to identify the list of interesting sensors. State management and other systems support are provided by exposing appropriate API, which we describe in the next section.

4 Mobile Virtual Sensor Layer

The Mobile Virtual Sensor (MVS) layer can be seen as an interface between the programmer and the smart surveillance system, which serves two goals. First, it raises the level of abstraction at which the surveillance application developers need to program. Instead of dealing with the physical sensors, application developers can express actions in terms of logical targets by instantiating, controlling and releasing MVSs. Secondly, the MVS layer allows smart surveillance system developers to provide the critical system properties such as scalability (through feature prioritization and resource allocation), reliability (through state replication, byzantine fault tolerance, and transparent MVS remapping), and security (through isolation and cross-validation). The challenge of designing the MVS layer consists in defining the minimal and complete set of primitives that satisfies both goals. In what follows, we describe the requirements of the API, identify the key architectural issues, and describe the open problems specific to the MVS Layer.

The MVS Application Programming Interface (API) defines how an application interacts with the Mobile Virtual Sensor (MVS) layer. In the common case, MVS remapping is reactive (MVS "follows", or is "attracted" by the target). In this case, the application can ignore the target mobility and how its tracking is achieved by successive MVS remappings to nearby physical sensors. For advanced programming, the API should allow application to explicitly control the MVS "moves" in advance. This is the case when a target's path can be anticipated (through machine learning techniques, for instance). Essentially, the MVS API primitives cover the following basic operations:

Starting: This portion of the API is used to specify the conditions that trigger an MVS to start delivering data to the application. From the application point of view, these conditions correspond to the detection of a "suspicious" event. Applications should also be allowed to issue direct commands to an MVS to start sending data to the application.

Tracking: This portion of the API is used to define the MVS sensing operations using its mappings to selected physical sensors. In other words, the API should allow programmers to indicate the physical data to collect and the set of sensors from which to collect the data. The application should be allowed to dynamically change MVS tracking operations (for instance, add another mapping to provide an additional audio stream) when certain conditions are met.

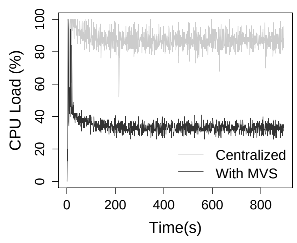
Moving: This portion of the API is used to specify the conditions that must be met for an MVS to change the set of physical sensors it is currently monitoring (through remapping). This also includes a mechanism for the application to issue direct commands to an MVS to move (this could be location-based or based on some other attribute) from some physical sensors to others.

Stopping: This portion of the API is used to specify the conditions that must be met for an MVS to stop delivering data to the application and terminate its execution. This also includes a way for the application to issue direct commands to an MVS to stop (or pause) the data stream sent to the application.

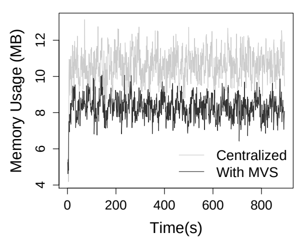
To support the described API, the MVS layer manages the MVS to physical sensor mapping starting from the physical layout of the sensors and the MVS targets. In doing this, the MVS layer implements various systems optimizations such as feature prioritization and resource scheduling. Below, we briefly describe the different mapping operations supported by the MVS layer:

Expanding: Since an MVS is dynamically mapped to a set of physical sensors, the MVS has to be able to identify (lookup/naming) new sensors it is interested in mapping to, to determine the resources these sensors have available, and to register the dynamic mapping between itself and the physical sensors. For shared sensors (with multiple MVS mappings), resource allocation and scheduling must be implemented.

Reducing: The complement of expansion is reduction. In



(a) CPU Load



(b) Memory Usage

Figure 4. Resource usage comparison in a camera network: cameras produce data at 5 fps with 320x240 resolution. Image processing and RFID reading happen on a 1.4GHz Pentium processor.

this case, an MVS must be able to remove the dynamic mapping between itself and a physical sensor, release any associated resources, and notify the other MVS's that it is no longer interested in sharing that physical sensor.

Merging: Since an MVS can be dynamically mapped to multiple physical sensors, it will have context (state) for each of these dynamic mappings. Furthermore, the state of one dynamic mapping may be continued later on in the context of a different dynamic mapping. For this to be possible, migration of MVS mapping state between physical sensors must be supported. When two mapping contexts of the same MVS migrate from two different physical sensors to the same sensor, there must be a way to merge them into a single context for the new mapping.

Preliminary Results: In order to quantify the benefits of selective attention, which is the underlying design principle behind the MVS layer, we did simple experiments in our camera sensor network testbed. A single object was being tracked in a network of 4 cameras, each camera producing data at 5 fps with 320x240 resolution. The object was tagged with RFID to detect the object's location. The location information was used to selectively process camera data for further analysis. Our experiments show 60% improvement in CPU usage because of selective attention. Similarly, overall memory usage in a system that uses selective attention is considerably less than a centralized system.

5 Related Work

The basic mechanism used to support the MVS layer architecture will leverage our previous work on Smart Messages [3], Spatial Programming [1] and Spatial Views [6]. Smart Messages represent a lightweight mobile agent platform that supports location-aware distributed computing through execution migration. DFuse, a system for distributed data fusion, uses a cost-function driven heuristic algorithm for dynamically mapping an application task graph on a physical network [7]. ASAP provides scalable resource management by using application-specific prioritization cues [8]. Other projects that are close to our proposed MVS architecture include the activity topology design based surveillance middleware [9] and the high level abstractions for sensor network programming paradigm [5, 4]. The middleware [9] approaches scalability by partitioning systems according to an activity topology describing the observed (past) behavior of target objects in the network. We plan use the activity topology to implement MVS movement logic in our system. EnviroSuite [5] and State-Centric programming [4] provide programming abstractions to hide the details of distributed applications such as distributed monitoring, tracking algorithms, collaboration groups, migration, and communications. In their examples of tracking application, the system migrates the leader to a neighbor node to follow a moving object. While their migration schemes are similar to our resource migration, the MVS layer considers object/state migration as well as resource constraints such as communication and computation in the presence of vision processing. Mobile agent based systems is another area of related research that is useful in supporting the mobility aspects of the MVS interface.

6 Summary

This paper explores the need and challenges involved in providing system support for real-world sensor-aided smart surveillance applications. To accomplish this goal, the paper proposes a system design alternative based on a novel mobile virtual sensor interface through which programmers can develop surveillance and target tracking applications. This new abstraction will remove the burden currently placed on developers in programming for multi-modal sensors with heterogeneous programming interfaces, as well as, for tracking targets. A reference implementation is currently being built to both validate the design choices and to evaluate the overall functioning (e.g., performance, features, programmability, scalability, etc.) of the system.

7 References

[1] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 690–699, 2004.

[2] https://www.buildingtechnologies.siemens.com. Video Surveillance Integrated Surveillance Systems.

[3] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *Comput. J.*, 47(4):475–494, 2004.

[4] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 2(4):50–62, October 2003.

[5] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3):543–576, 2006.

[6] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. *SIGPLAN Not.*, 40(6):249–260, 2005.

[7] U. Ramachandran, R. Kumar, M. Wolenetz, B. Cooper, B. Agarwalla, J. Shin, P. Hutto, and A. Paul. Dynamic data fusion for future sensor networks. *ACM Trans. Sen. Netw.*, 2(3):404–443, 2006.

[8] J. Shin, R. Kumar, D. Mohapatra, U. Ramachandran, and M. Ammar. ASAP: A camera sensor network for situation awareness. In *OPODIS'07: Proceedings of 11th International Conference On Principles Of Distributed Systems*, 2007.

[9] A. van den Hengel, A. Dick, and R. Hill. Activity topology estimation for large networks of cameras. In *AVSS '06: Proceedings of the IEEE International Conference on Video and Signal Based Surveillance*, page 44, Washington, DC, USA, 2006. IEEE Computer Society.