A Distributed Framework for Spatio-temporal Analysis on Large-scale Camera Networks

Kirak Hong*, Marco Voelz†, Venu Govindaraju‡, Bharat Jayaraman‡, and Umakishore Ramachandran*
*Georgia Institute of Technology
{khong9, rama}@cc.gatech.edu
†University of Stuttgart
marco.voelz@ipvs.uni-stuttgart.de
†SUNY Buffalo
{govind, bharat}@buffalo.edu

***Abstract*—Cameras are becoming ubiquitous. Technological advances and the low cost of such sensors enable deployment of large-scale camera networks in metropolises such as London and New York. Applications including video-based surveillance and emergency response exploit such camera networks to detect anomalies in real time and reduce collateral damage. A well-known technique for detecting such anomalies is spatio-temporal analysis – an inferencing technique employed by domain experts (e.g., vision researchers) to answer spatio-temporal queries.**

Performing spatio-temporal analysis in real-time for a large-scale camera network is challenging. It involves continuously analyzing the images from distributed cameras to detect signatures, generating an event by comparing the detected signature against a database of known signatures, and maintaining a state transition table that show the spatio-temporal evolution of people movement through the distributed spaces. Being inherently distributed, computationally demanding, and dynamic in terms of resource requirements, such applications are well-positioned to exploit smart cameras and cloud computing resources. However, developing such complex distributed applications is a daunting task for domain experts.

In this paper, we propose a distributed framework to facilitate the development and deployment of spatio-temporal analysis applications on large-scale camera networks and backend computing resources. The framework requires the domain experts to provide a set of handlers that perform the domain-specific analyses (e.g., signature detection, event generation, and state update). The runtime system invokes these handlers automatically in the distributed environment consisting of smart camera networks and cloud computing resources. We make the following contributions: (a) a distributed programming framework for spatio-temporal analysis, (b) a careful investigation of the computation/communication costs associated with the large-scale spatio-temporal analysis to arrive at the scalable system architecture, (c) automatic resource configuration to cope with the dynamic workload, (d) a detailed performance evaluation of our system with a view to supporting scalability and quality of service.

***Keywords*-distributed programming framework; resource management; runtime systems; camera networks; video-based surveillance; spatio-temporal analysis;**

I. Introduction

As sensors for recognizing humans, such as cameras, voice recognition sensors, and RFID readers, are becoming more capable and widely deployed, new application scenarios arise, requiring an automated processing of the continuous stream data to identify and track human beings in real-time. Applications in this domain include airport security, emergency response and assisted living, all requiring real time detection of unusual situations, called *anomalies*. Different from techniques such as RFID badges, cameras allow for an unobtrusive way of identifying people's whereabouts, making them the primary source of information in many of these scenarios.

Take an airport scenario as an example: Amsterdam's Schiphol airport currently has 1,000 cameras in place and plans to increase that number to between 3,000 and 4,000 over the next few years [1]. In an airport, a common security violation is that an individual enters into a restricted area without permission. If such a situation happens, the individual should be reported to an airport security team in real time, preventing potential threats to the airport.

The high level goal in such applications, often referred to as *situation awareness* applications [2], is catching anomalies in real time and reducing collateral damage. To achieve this, there is a well-known technique called *spatio-temporal analysis*, enabling an application to answer spatio-temporal queries on known occupants such as "Where is person A?", "When and where did person A leave zone X?", "When and where did person A and person B meet for the last time?".

Applications providing the means to answer these queries usually employ distributed cameras and sensors of other modalities (such as audio and biometrics) to detect people in the observed system. These live sensor streams are used to make an estimation about the identity of the detected people, comparing the data to a set of well-known identities. These estimates generated throughout the system are gathered and regularly consolidated to create a global view of the observed area, e.g., by recording the most likely whereabouts of each person known to the system at a certain point of time. The current global state and possibly a history of former states enable the system to answer queries such as stated above.

Recently, Menon et al. [3] showed the feasibility of spatio-temporal analysis using this concept by maintaining the *global state* in a *transition table* similar to hidden markov models. The table represents the probabilities of each occupant known to the system being in each of the observed zones. *Events*, which indicate that an occupant has been observed within a zone, trigger a transition from the current state to the next. Just as the global state, events are represented by probabilities rather than exact knowledge, because algorithms for signature detection and comparison are inherently inaccurate.

To develop a large-scale situation awareness application using the spatio-temporal analysis technique, questions of system scalability and efficient resource management will arise and must be addressed. In large settings such as airports or urban environments, processing the data streaming continuously from multiple video cameras is both data- and compute-intensive task. Moreover, real-time situation awareness applications have latency-sensitive quality of service requirements, while their workloads are highly dynamic depending on the situation of the physical environment (e.g., level of activity in an airport). Developing a large-scale distributed application to meet such requirements is a daunting task for domain experts. Questions of system scalability go beyond video analytics, and fall squarely in the purview of distributed systems research.

In this paper, we develop a distributed programming framework to enable domain experts to develop large-scale situation awareness applications using spatio-temporal analysis technique. Specifically, we make the following contributions:

- We develop a distributed framework for real-time spatio-temporal analysis that can be used by domain experts to "plug and play" their algorithms.
- We investigate performance bottlenecks in large-scale spatio-temporal analysis and design a distributed system architecture to support scalability and quality of service.
- We provide automatic configuration of backend system resources to satisfy quality of service with given application parameters and dynamic workloads.
- We implement the framework, and evaluate scalability and quality of service of the proposed system with realistic domain-specific algorithms.

Section II presents other work related to our project. Section III discusses application logic of spatio-temporal analysis and challenges of performing such applications on large-scale camera network to arrive at the design decision for the scalable system architecture. Section IV explains details of the framework including its programming model and resource management, as well as support for scalable state update. Section V gives implementation details of the framework, including resource reconfiguration protocol and a mechanism to ensure temporal order of events. Section VI provides performance measurements of the implemented system and show scalability of the proposed approach. Section VII presents concluding remarks and future work.

II. Related Work

There are many middlewares and frameworks that help developing applications for smart environment and situation awareness applications. For example, the EasyLiving project [4] presents a software architecture for smart environments that includes a person tracking system based on color stereo for maintaining the identities and locations of people. Gaia meta-operating system [5] provides a framework for building user-centric applications in active spaces, where users seamlessly interact with their surrounding physical and digital environments. SLIPstream [6] presents a scalable programming framework that allows interactive perception applications to run on distributed nodes to process streaming data with low latency. However, the focus of such systems is user-centric pervasive applications in room or building scale smart environments; our focus is large-scale camera networks and the application focus for us is real-time spatio-temporal analysis, quite different from these other systems. IBM S3 [7] provides a middleware for video-based smart surveillance system including Smart Surveillance Engine (SSE) for the front end video analysis capabilities and Middleware for Large Scale Surveillance (MILS) for data management capabilities. However, it does not provide a domain-specific programming model that is one of the primary contribution of our work. Target Container [8] is a parallel programming model for video-based surveillance applications, which allows domain experts to easily write a large-scale surveillance applications. However, it does not address the problem of handling dynamic workload from large-scale camera network, which is one of the biggest challenge in developing a situation awareness application on a large-scale camera network.

There has been extensive research in the area of video analytics and image recognition that are the enabling technologies for spatio-temporal analysis. For example, robust real-time face detection algorithms [9], [10], [11] can extract faces from video frames. There also exist many algorithms that can accurately recognize an identity by comparing a face image to known faces [12], [13], [14], [15], [16] or using human gaits [17], [18]. BioID [19] uses multi-modal biometric information such as face, lip movement, and voice for person identification in order to achieve higher accuracy than using a single biometric data. While domain experts have developed robust algorithms and techniques for analyzing video and exploiting multi-modal biometric information for

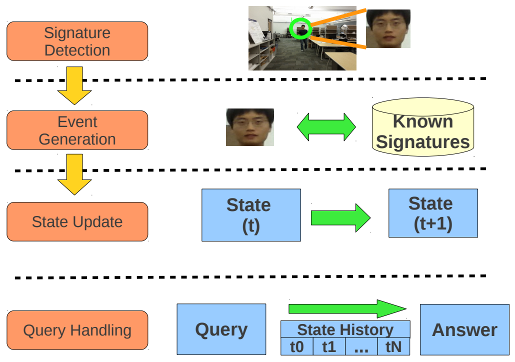


Figure 1. Application structure for spatio temporal analysis: This is a general structure of spatio-temporal analysis on camera network that answers target locations at given time using camera streams.

enhancing the quality of spatio-temporal analysis, applying such techniques to a large-scale distributed camera network consisting of 1000s of cameras is beyond the expertise of such domain experts. Developing such a framework is the focus of this paper.

The system we develop and implement is intended to A domian expert of situation awareness application using such advanced biometric

III. Design Space Exploration for Large-scale Spatio-temporal Analysis

In this section, we discuss the common steps in spatio-temporal analysis and identify potential bottlenecks for large-scale spatio-temporal analysis. To solve the bottlenecks, we analyze each step's workload characteristic and make a design choice for a system architecture consisting of a smart camera network and elastic computing infrastructure to enable spatio-temporal analysis for a large-scale camera network.

A. Application Logic

In general, spatio-temporal analysis enables an application to answer queries referring to locality- and time-dependent information about different occupants. Common examples of spatio-temporal queries include:

"Where was person A at time T?"
"When did person B leave zone X?"
"When and where did person A and B meet for the last time?"
"Who moved from zone X to zone Y between time T1 to T2?"

To answer these queries, an application has to maintain its state which represents each occupant's location at different point of time. Figure 1 shows a general application pipeline of spatio-temporal analysis involving four steps: Signature detection, event generation, state update, and query handling.

Signature detection involves video analytics to detect signatures such as faces. For example, when a person enters a zone, a face detection algorithm reports the person's face by analyzing video frames from a camera observing the zone. Note that multiple signatures can be reported from a single frame. For each frame, the face detection algorithm finds all image regions containing faces and passes them to the event generation step.

Event generation involves generating a probabilistic estimate about the identity of a detected signature. Depending on the application, different algorithms can be used in this step. For example, various face recognition algorithms [20] or human gait recognition [17], [18] may be used to generate an event, which includes similarities between the detected signature and known signatures.

State update maintains an application-specific state based on the observed events. The goal is to reflect in the global state the information provided by an event, e.g., that Person A was seen in Zone 2 with a probability of 0.75. The state of an application represents its knowledge about each occupant's location at a given time. A new event causes an update from one state to another, using the new information about a specific occupant's location. Different state update algorithms can be used depending on the application's information needs. For example, Menon et al. [3] proposed a simple state transition function that increases probabilities of an occupant being in a zone proportionally to the similarities between the detected signature and known signatures. More complex algorithms may be used, such as one taking adjacent zones and possible paths across zones into account for better accuracy.

Query handling uses the current and past application states to answer various spatio-temporal queries. Although it is an essential step for situation awareness applications, we do not consider query handling for the system design since it is not in the critical path of real-time event processing. In this work, we focus on signature detection, event generation and state update. APIs and mechanisms for efficient query handling is part of our future work.

B. Qualitative Analysis of the Workload for Large-scale Spatio-temporal Analysis

In large-scale spatio-temporal analysis, each processing step becomes a potential bottleneck due to their computation and communication costs. First, large volume of input video streams and computationally intensive video analytics makes signature detection a bottleneck. For example, transmitting thousands of video streams into a centralized server and performing video analytics on the streams in real time is not feasible. Second, event generation becomes another bottleneck since potentially a large number of signatures would be generated simultaneously, each of which involves computationally intensive comparisons to a large set of known signatures. Similarly, state update becomes a bottleneck due to the large number of events generated where each event causes updating a large application state.

Step	Number of inputs	Size of each input	Parallelism
Signature detection	static	large	Different parallel streams
Event generation	dynamic	small	A detected signature is compared against a common database of known signatures to generate an event. A number of such independent events may be generated in parallel.
State update	dynamic	small	Calculation of probability of occupants in different zones. Each event affects the global state of the occupants in different zones. Need to preserve the temporal order of the events in updating the global state.

Table I
WORKLOAD CHARACTERISTICS OF DIFFERENT STEPS IN SPATIO-TEMPORAL ANALYSIS

To solve the bottlenecks, it is necessary to distribute the workload of large-scale spatio-temporal analysis among distributed computing nodes taking into account the workload characteristics of each step. Table I shows the different workload characteristics for different steps of the spatio-temporal analysis. Signature detection involves the processing of large amounts of video data from the distributed cameras. The number of input streams is a function of the actual physical deployment of the camera network, which could evolve over a long period of time but does not change in the short run. Plus, signature detection on different video streams can be performed in parallel since they are independent computations. The size of the input data for event generation is relatively small, namely, the size of a signature such as a ”face”, which is much smaller than a raw video frame needed for signature detection. However, the number of events being generated in parallel is highly dynamic and depends on the number of detected signatures from the camera streams. For example, number of faces detected from a camera stream in an airport during the day is typically much larger than at night. Event generation for each detected signature can be performed in parallel since each event generation represents an independent processing step. The input to the state update step is an event. Each event is a set of values representing probabilistic estimates of a specific occupant being in different zones, which is typically much smaller than a raw image. The number of events is also highly dynamic since each detected signature leads to the generation of an event. Unlike the previous two steps, state update requires sequential processing of events, preserving the temporal order of events. This is because the processing of each event can potentially change the relative probabilities of all the occupants in all the zones. However, the state update step offers ample opportunities for exploiting parallelism for updating the application state. Details of how such parallelism of different steps are exploited in our framework will be discussed in the Section IV-A.

C. Design Choice for Large-scale Spatio-temporal Analysis

In this section, we make a design choice for a system architecture, based on the workload characteristics discussed in the previous section. To allow real-time processing of signature detection on a large-scale camera network, it would make sense to use embedded smart cameras. Each smart camera processes its video stream in real time, sending only signatures when detected. Using smart cameras would significantly improve scalability of the system because large volume of videos are locally processed and only interesting signatures are reported to the system. For example, a stream of 640x480 Motion JPEG video at 30 frames per second requires 5 to 10 Mbps for streaming [21] while actual signature (e.g., a JPEG image of a face) that is meaningful for spatio-temporal analysis is only tens of killobytes. Plus, streaming all videos at all time, including unchanging videos, is wasteful since they do not contain any useful information.

Unlike signature detection, event generation is not well suited on embedded smart cameras since the computational cost is highly dynamic; an event generation overhead involves per-signature computing cost where number of signatures in a video stream changes over time. Overprovisioning resources for smart cameras is tricky since it is hard to make an assumption for the number of signatures detected. If such an assumption is violated, e.g., more signatures are detected due to the changes in physical environment, quality of service of the application will degrade due to the insufficient computing resources for real-time event generation. To overcome such a problem, elastic backend computing infrastructures (*a la* cloud) are good candidates because computing resources can be instantiated on demand. Moreover, each event generation is an independent computation, which makes this processing step linearly scalable on the elastic computing infrastructure.

Similar to signature comparison, state update also has dynamic workload and therefore elastic backend computing resources are attractive to reduce this bottleneck. Plus, state update has a hidden communication cost due to the partitioning and distribution of an application state over multiple nodes. Typically such communication cost is much cheaper in the elastic computing infrastructure than in the smart camera network. More details about the communication cost for state update will be discussed in Section IV-C.

IV. SYSTEM ARCHITECTURE

In this section, we discuss details of our destributed framework, including programming model, automatic resource configuration, and system support allowing selective heuristics for scalable state update. Figure 2 shows our framework to support spatio-temporal analysis on large-scale camera networks. The framework provides a domain specific

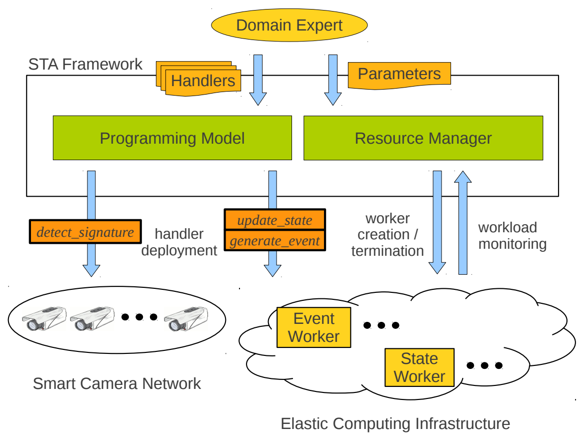


Figure 2. Conceptual view of our framework: The framework consists of programming model and resource manager. The programming model helps domain experts in developing large-scale spatio-temporal analysis applications, while the resource manager takes care of dynamic deployment in the cloud.

```
//application-specific handlers
Signature[] detect_signature(VideoFrame);
EventElement[numOccupants] generate_event(
    Signature);
StateElement[numZones] update_state(
    OccupantID, StateElement[numZones]);
```

Figure 3. Handlers in our programming model: Three handlers are provided by domain experts.

programming model allowing the domain expert to develop the application as a set of code modules called *handlers*. The handlers implement the steps of the spatio-temporal analysis needed in the application; the handlers are automatically invoked by the runtime system backing the programming model. The framework includes a resource manager that automatically configures the distributed computing nodes in the elastic computing infrastructure adjusting to the workload dynamics to perform spatio-temporal analysis in real time.

A. Programming Model

Figure 3 shows the application-specific handlers to be provided by the domain experts and registered with the runtime system using our programming model. The domain expert provides three handlers: There are three handlers in our programming model: *detect_signature*, *generate_event*, and *update_state*. Once registered, each handler is invoked automatically by the runtime system to work on the corresponding input data. For example, *detect_signature* handler is invoked when a new frame is available while *generate_event* is invoked when a new signature is reported.

Figure 4 shows the roles of the handlers, along with thier input and output data. The role of the *detect_signature* is to analyze each camera image it receives to detect signatures. A domain expert would code up a video analytics algorithm in this handler that detects application-specific signatures such

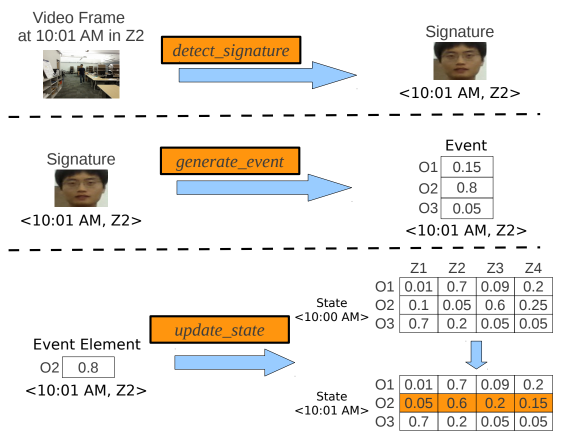


Figure 4. Roles of the handlers in the spatio-temporal analysis application. The programming model invokes these handlers automatically upon the availability of input data.

as faces and human gaits. The handler returns an array of detected signatures where each signature is automatically tagged with the current wall clock timestamp and zone ID of the camera by the runtime system backing the programming model. The runtime system also guarantees that the handler is invoked sequentially for images from a single camera stream. This allows the domain expert to develop a stateful video analytics algorithm in the handler (such as adaptive algorithms that distinguish background and foreground [22], [23] to avoid detecting face-shaped background objects as faces).

Once a signature is captured, the *generate_event* handler is invoked. This handler generates a single event upon processing the detected signature. The runtime system automatically tags the generated event with the timestamp and zone ID derived from the input signature. Thus the programming model automatically provides propagation of temporal causality in the spatio-temporal analysis application.

An event is an array of an application-specific data, where each element of the array is associated with a known occupant ID. For example, *generate_event* algorithm could be a face recognition algorithm that compares the signature (a detected face) to known faces, generating an event that contains the application-specific similarity metric of the detected face to the known faces. Intuitively, a single event generation seems to be parallelizable since different comparisons between the detected signature and the known signatures are independent of one another. However, an event generation algorithm may have an arbitrary structure including sequential code, which makes it tricky to automatically parallelize the handler execution. For example, the Eigenface [12] algorithm uses *Principal Component Analysis* to transform a face image to an eigenface before comparing different eigenfaces. In fact, PCA takes most of the execution time in generating an event, which makes it less attractive to parallelize just the comparison part of

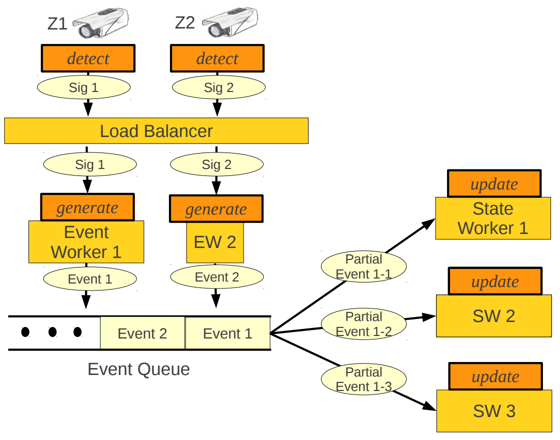


Figure 5. Execution of handlers on distributed nodes: Handlers are invoked on distributed smart cameras and worker nodes in the cloud. Hander names are abbreviated.

the algorithm. Other algorithms may want to normalize similarities between the detected signature and the known signatures [24], which involves a sequential process after the comparisons are completed. For these reasons, and to keep the programming effort of the domain expert simple, we do not attempt to parallelize a single event generation. In other words, *generate_event* handler is a sequential algorithm. However, the event generation for each detected signature can be executed in parallel. Therefore, we exploit parallelism at the level of multiple event generations, instead of different comparisons in a single event generation.

For each generated event, the application state should be updated to show the temporal evolution of occupants in different zones. Figure 4 shows how the *update_state* handler updates an application state from the current state to the next state. The application state is a table (two-dimensional array) of application-specific data indexed by occupants and zones. In this table, each row is called an *occupant state* since it gives the information of the whereabouts for a specific occupant; each column is called a *zone state* since it gives the information about the known occupants in a specific zone. As should be evident, each occupant state is independent since movement of different occupants are independent. However, elements in a single occupant state are coupled. For example, a high probability in a specific zone would result in low probabilites in other zones. Based on this observation, the *update_state* handler is designed to update a single occupant state upon invocation (Figure 4), allowing our framework to exploit the inherent parallelism of state update. For a single event, a set of *update_state* handlers can be invoked in parallel, on different occupant states with different elements of the event (Figure 4). This makes it possible to distribute the workload of state update over distributed computing nodes.

Once handlers are registered, they are invoked by the runtime system on each of the distributed nodes. Figure 5 shows the execution of the handlers on the distributed nodes. The *detect_signature* handler is invoked for each video frame at the distributed smart cameras. If and when a signature is returned by *detect_signature*, it is delivered to any available worker node in the cloud, called an *event worker*. Different signatures detected from a single smart camera can be delivered to different event workers in order to achieve load balancing across distributed event workers. At each event worker, *generate_event* handler is invoked with the input signature to generate an event. While events are independently generated at different event workers, the events are globally ordered based on their timestamps. Respecting this global temporal order, different elements of an event are delivered to specific distributed worker nodes called *state workers*.

Each state worker maintains a specific occupant state; it invokes the *update_state* handler with the input event element to update its specific occupant state. Note that the programming model is flexible and allows a state worker to be responsible for multiple occupant states. The important point to note is that this structure allows parallel invocation of *update_state* on the different state workers. More details about the implementation issues for achieving load balance and global temporal order of events are discussed in the Section V.

B. Automatic Resource Management

One of the key *quality of service* metric of importance to the application in spatio-temporal analysis is the *latency* from signature detection to state update. As we have already observed, the workload in this application is highly dynamic. The worst case rate of event generation is the product of the frame rate of the smart cameras and the number of signatures detected in each frame. As long as each smart camera has enough processing power to do the signature detection in real time keeping up with the frame rate of the camera, the signature detection step of the application will not be a problem in meeting the quality of service metric. The computational resources needed to do the event generation and consequently the state update in response to each event changes dynamically with the rate of detection of signatures by the smart cameras. While it is conceivable to over-provision the computational infrastructure to meet the worst case event generation rate, such a strategy will obviously be wasteful of the computational resources. Therefore, it is necessary to manage the computational resources to meet the quality of service requirements without undue over-provisioning of the resources.

To address this issue, our framework provides a resource manager using the elastic computing infrastructure of the cloud. The solution approach we have taken is to require the domain expert to provide a minimal set of parameters that can be used by the resource manager to automatically tune the resource configuration to meet the QoS needs of the application.

There are three steps involved in such an automatic resource management. First, we perform off-line profiling to generate configuration lookup tables that can be used at runtime to decide the "right" configuration to use for event generation and state update commensurate with the dynamically changing workload and the QoS requirement. for event generation and state update, which includes configurations for each level of workload and latency requirement. Second, at application start-up, we make an assumption about the initial workload and choose a configuration from the configuration lookup table based on the assumed workload and user-provided latency requirement. Third, once an application is up and running, the resource manager adaptively changes the resource configuration in order to satisfy the quality of service requirement with changing workload.

Profiling: For the off-line profiling, the following items are provided by a domain expert: a set of handlers, number of zones and occupants in the system, maximum event rate, and a dollar figure for the budget for the resources. Given these inputs, the resource manager profiles the event generation and state update steps separately for all configurations within the specified budget and the event rates within the maximum event rate, to build the configuration lookup table. The profiling involves three control variables: the types of workers, the number of workers, and the event rate. The event rate for state update is equal to the signature detection rate, and both are emulated by a workload generator that provides sample signatures and events at a specific rate. For each profile run, the resource manager measures the latencies for event generation and state update over time. The profiler ignores configurations in which the latencies keep increasing with time. Each valid configuration with stable latencies is recorded in the configuration lookup table, where configurations are indexed by the average latency and event rate. This table is used by the resource manager at runtime, to change a configuration for a specific level of workload while satisfying a given latency requirement.

Initial Configuration: Once the profiling step is completed, we are ready to launch the spatio-temporal analysis application. For the launch of the application we have to make an assumption about the initial workload to configure the resources. Since the latency is crucial for most applications in the domain, the initial configuration is picked based on the maximum event rate. Given the maximum event rate, the resource manager finds a configuration satisfying the latency requirement from the configuration lookup table. If multiple configurations are found, it chooses the one with the minimum cost.

Runtime Adaptation: Once an application starts to run, the resource manager adaptively changes the resource configuration based on the current workload. To estimate the current event rate, that is directly related to the level of workload, the resource manager broadcasts a message to all event workers for each period and gathers the number of events generated since the last period. Based on the estimated event rate, the resource manager decides whether to increase or to decrease the number of workers if necessary.

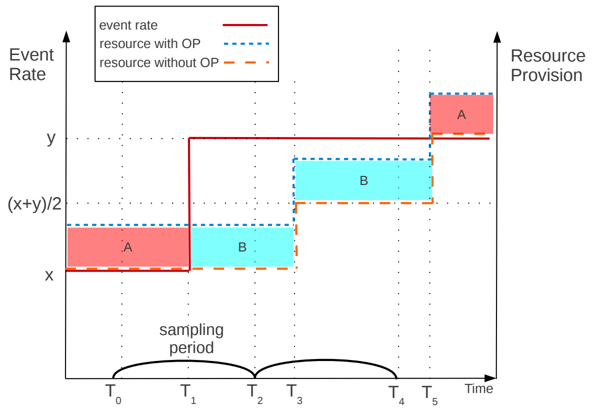


Figure 6. Resource adaptation: Resource manager adapts resource configuration when the estimated workload has been changed.

Figure 6 shows the example scenario of such an adaptive resource configuration. In the figure, there are two y axes, one for event rate and another for resource provision. All lines can be interpreted both in terms of event rate and resource provision, since they are directly related. The solid line means the actual workload and the coarse dashed line shows the resources provisioned without overprovisioning, while the fine dash line indicates the resources provisioned with overprovisioning. At T_0, the event rate is x and there are enough resources to handle the incoming events in real time. At T_1, the workload is suddenly changed from event rate x to y. However, the resource manager does not recognize the workload change yet since it is in the middle of a sampling period. From this point (T_1), the quality of service starts degrading, since more events are coming than that can be handled in real time. At T_2, the changed workload is reported as event rate $\frac{(x+y)}{2}$, since the event rate was x for the half of the last period. Based on the estimated event rate, the resource manager makes provision to more resources to handle the increased workload. Although resources are provisioned, there can be a delay until the resources become available, depending on the elastic computing infrastructure used. For example, computing instances in Amazon EC2 need to finish the boot up process of operating systems before available. After the delay (T_3), the provisioned resources are available and therefore the higher event rate can be handled. However, the resources are still not enough to handle the actual workload, since the provision was based on the misestimated event rate $\frac{(x+y)}{2}$. At T_4, the resource manager finally estimates the actual event rate y, and makes a provision to more resources to handle the workload. Again,

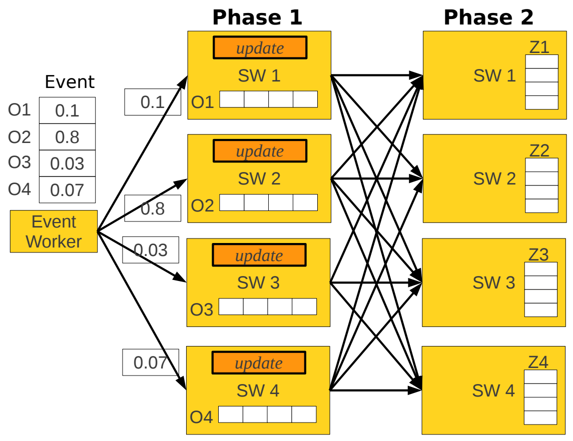


Figure 7. Distributed state update: For each event, occupant states are updated in parallel at distributed state workers by invoking *update_state* handler (Phase 1). Once all occupant states are updated, the elements of occupant states are exchanged among state workers to update zone states (Phase 2).

after a certain delay (T_5), the provisioned resources become available and the quality of service will gradually recover.

As shown in Figure 6, overprovisioning has an advantage in terms of quality of service and an disadvantage in terms of cost. When resources are overprovisioned, a certain amount of resources are wasted when there is no workload change (regions makred with 'A'). However, when the system experiences a sudden workload increase (at T_1), overprovisioned resources makes less quality of service degradation by using the overprovisioned resources (regions marked with 'B').

There are three parameters that affect the quality of service degradation during the resource adaptation: amount of overprovisioning, resource provision delay, and workload sampling rate. Resource provision delay is a delay from resource provisioning to the point when the resources become available. The delay typically includes boot up process of operating systems when using IaaS cloud such as Amazon EC2. The resource provision delay is a parameter specific to the elastic computing infrastructure, and therefore it cannot be changed by our framework. The amount of overprovisioning can be changed by a domain expert since some applications are more critical than others, which makes large amount of overprovisioning a reasonable choice. The workload sampling rate is a parameter that is related to frequency of workload changes. If the sampling period is small, it becomes sensitive to small workload changes, while larger sampling period will not affected by the small workload changes.

C. *Selective State Update for Scalability*

Our framework employs distributed nodes for the different steps of the processing involved in spatio-temporal analysis. Of the three steps, there is no scalability issue with signature detection which happens in the smart cameras. This is based on the reasonable assumption that there is sufficient computational power in the smart cameras to deal with the frame rate of a single camera stream. Each individual event generation step is independent of other event generation steps that may be going on in parallel. Thus the event generation step also does not pose any problem from the point of view of scalability with the adaptive resource management that ensures that there is sufficient number of event workers to deal with the dynamic load. State update step is the one step that requires careful orchestration to ensure scalability. We explain this step in more detail in this section.

Figure 7 shows the distributed state update using multiple state workers. Each state worker is responsible for a specific set of occupant states and zone states. For example, State Worker 1 maintains the occupant state of O1 and the zone state of Z1. The occupant states and zone states are essential to answer spatio-temporal queries. The occupant state captured via the rows, allows the spatio-temporal framework to answer a query such as "where is occupant O1?", or "what is the the track of occupant O1 for the past 10 minutes?". On the other hand, the zone state captured via the columns allows the framework to answer queries regarding a specific zone such as "who are presently in zone z1". When a new event is generated, different elements of the event are delivered to different state workers regarding commensurate with the occupant states that each is responsible for.

As shown in Figure 7, upon the arrival of an event element, the *update_state* handler for the specific occupant state is invoked at a state worker. Once the occupant states are updated, elements of occupant states are exchanged among the state workers to update the zone states (phase 2). Note that phase 2 only involves data copy and not any new computation. As shown in the figure, computational workload of state update is distributed over state workers in phase 1. However, each state worker has to communicate to all other state workers to update the zone states in phase 2. We need more state workers to reduce the latency for state update for each event. One possibility is to simply stop with phase 1 since the state update is complete at this point. However, this will make query processing inefficient since zone-specific queries will need to contact *all* the state workers. This is the reason for making each state worker responsible for a set of occupants and a set of zones. Thus the communication in phase 2 to update the zone states in all the workers is a necessary evil for efficient query processing. The more the state workers the more will be the communication in phase 2. This has the undesirable effect of increasing the latency for state update, an equally important quality of service metric for such applications.

To achieve better scalability, applications may choose to selectively update application states. For example, the event generation algorithm by Menon et al. [3] gives only few non-negligible comparison results when realistic face images are used. If an event generation algorithm is highly accurate,

```
//API for selective state update
void set_occupant_mask(set<OccupantID>);
void set_zone_mask(set<ZoneID>);
```

Figure 8. API for selective update

i.e., giving high probability for the ground truth identity and very small probability for other identities, the application can effectively apply a threshold and provide only meaningful event elements for updating specific occupant states. Similarly, a state update algorithm can threshold negligible changes of an occupant state, providing only meaningful state elements for updating specific zone states.

To support such selective update heuristics, we provide two API calls in our framework as shown in Figure 8. The *set_occupant_mask()* can be used in the *generate_event* handler in order to mask out negligible event elements. The masked event elements will not be transmitted to state workers during phase 1 of the state update step. The *set_zone_mask()* can be used in the *update_state* handler to mask negligible changes in an occupant state. The occupant state elements that are masked will not be transmitted to other state workers during phase 2 of the state update step.

The impact of such heuristics on system performance depends on the selectivity of the heuristics, which may be linked to the algorithmic accuracy of event generation and state update. For example, a single message for an event element is enough if an ideal event generation algorithm generates a definite answer: true for the ground truth identity and false for all other identities. Ideal state update algorithm would need two messages at a maximum, one for the zone from where an occupant has disappeared, and one for the zone where the occupant has appeared. However, such ideal cases are unlikely; most event generation and state update algorithms provide probabilistic estimate about the identities and locations. If the estimate is too vague (e.g., uniform probabilities for all identities or all zones), more messages are required since all elements are non-negligible. This means that poor algorithms in terms of accuracy may result in poor system performance as well. The accuracy of the algorithm is in the hands of the domain expert and not relevant to this study. We focus on the performance behavior of our framework when applications can selectively update different amount of application states.

V. Implementation

In this section, we discuss implementation details of our framework, particularly focusing on resource reconfiguration using Amazon EC2 and event ordering for state update. The current version of our framework implementation uses Amazon EC2 as an elastic computing infrastructure, although the design of our framework is not limited to a particular infrastructure.

A. Reconfiguration of Computing Resources in the Cloud

The resource manager in our framework dynamically adds and removes computing resources at runtime. To use the on-demand computing resources for event workers and state workers, our framework prepares two Amazon Machine Image (AMI), one for the event workers and one for the state workers. AMI is a virtual machine image which is used to create a virtual machine within the Amazon Elastic Compute Cloud (EC2). AMI for each type of worker has a startup process that starts a runtime system that registers itself to the distributed framework, manages communication to other workers and smart cameras, as well as invoking handlers upon arrival of intput data.

Event Worker Pool: When new event workers are created by the resource manager, they register themselves to the *event worker directory*, a well known name server, after the boot up process of operating system. Each smart camera periodically requests for the updated list of event workers from this nameserver. If new event workers are returned from the event worker directory, the smart camera initiates tcp connections to the new event workers. This is to avoid the connection set up cost when a camera chooses to send its detected signature to a specific event worker. These tcp connections are kept alive until the event workers terminate. When a signature is detected from a smart camera, the signature is transmitted to an event worker that is randomly picked from the set of active event workers. When the resource manager decides to reduce the number of event workers (as part of its dynamic resource management strategy to save costs), the resource manager removes the associated event workers from the event worker directory and terminates the event workers. When the event workers are terminated, the smart cameras are notified for the closed tcp connections, which ensures that the cameras will not use the terminated event workers.

State Worker Pool: When new state workers are created by the resource manager and become functional after the boot up process, they register themselves to the *state worker directory*, a nameserver maintaining a list of active state workers as well as a *state lookup table* that maps state workers to particular occupant and zone states for a certain time range. When a new state worker is registered, the state worker directory adds the new state worker into the list of active state workers. The new state worker is assigned a portion of the occupant and zone states for future evolution of the state table from this time onwards. This information is entered into the state lookup table. Note that we do not migrate past application states to the newly incarnated state worker to minimize the reconfiguration cost. Once this initialization steps are completed, the information regarding the new state worker (and the occupant and zone states it is responsible for) is broadcast by this state nameserver to all other event and state workers. When some state workers are

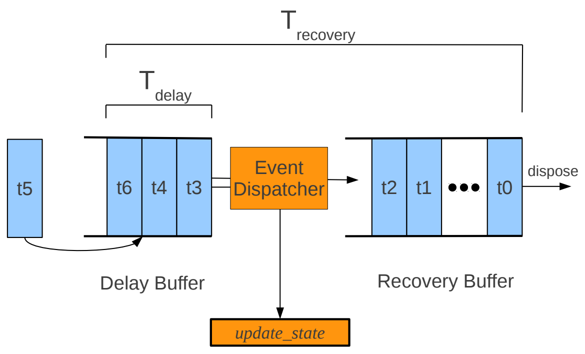


Figure 9. Event buffering: to ensure time-ordered state update, events are buffered in ordering buffer before being used by update handler. If an event arrives out-of-order after the state update has been done, the rollback buffer is used to recalculate the new application state.

to be terminated, the resource manager sends messages to the closing state workers, so that they migrate their occupant and zone states to other state workers that will survive after the reconfiguration.

B. Event Ordering for State Update

In many application scenarios, state update should preserve the temporal order of events. For example, suppose occupant A entered to zone X followed by occupant B entering zone Y. To preserve application correctness, it is necessary to perform the state update for event A before event B at every state worker to whom both the events are communicated. However there is no way to guarantee such ordering, given the vagaries of the network and the fact that our middleware has no control on the scheduling of the virtual computational resources allocated to an application on the physical data center CPUs. To solve this problem, we have developed a delay-and-recovery mechanism using local *event buffers* at the state workers.

Figure 9 shows our approach using two buffers: a *delay buffer* and a *recovery buffer*. We assume that all state workers and smart cameras have synchronized wall clock times using well known protocols such as Network Time Protocol [25]. There are two system parameters used in our approach: T_{delay} and $T_{recovery}$. T_{delay} is the latency bound used for delaying events before state update ; the hope is that most events would be delivered to the state workers from the event workers within T_{delay} such that the chances of state update happening out-of-temporal-order is fairly slim. However, there could always be stragglers in a large-scale distributed setting. To account for such stragglers, we buffer the output of the state update for a period $T_{recovery}$. Our system guarantees that the evolution of the state is accurate so long as all stragglers arrive within the $T_{recovery}$ time period. We explain our solution in the following paragraphs.

State Update: When an event element arrives at a state worker from an event worker, the timestamp of the event element is checked. If the event elements are delivered within the T_{delay}, i.e., the timestamp is in between the current time and the current time minus T_{delay}, the event elements are buffered in a delay buffer respecting the correct temporal order. The processing of an event element is delayed within this buffer until the gap between its timestamp and the current wall clock time is larger than T_{delay}. After such time, an event is dispatched by an event dispatcher, which continously monitors the delay buffer for the event elements that are ready to be dispatched. The event dispatcher invokes a set of *update_state* handlers with the event elements, respecting the temporal order of event elements for the same occupant state. Note that the event elements for different occupant states can be processed in parallel regardless of the temporal order. Once the event elements are processed, they are placed into recovery buffer.

State Recovery: If a new event element arrives out of temporal order (i.e., events with later timestamps have already been processed), then the application state recovery is triggered to recover the corresponding occupant state. When the recovery procedure is triggered, event dispatcher is stopped to avoid any side effect. To recover the occupant state, *update_state* handler is sequentially invoked with all event elements for the occupant state from the point where temporal order is broken[1], including the newly arrived event element. The cost of the recovery procedure depends on the number of event elements already processed, which in turn, is related to how much the estimated latency bound (T_{delay}) is violated. Once event elements become very old, i.e., the gap between their timestamps and the current wall clock time become larger than $T_{recovery}$, the event elements are removed from the recovery buffer. This means that the application state older than $T_{recovery}$ cannot be recovered.

For spatio-temporal queries regarding time between $T_{current}$ and $T_{current} - T_{recovery}$ where $T_{current}$ is the current wall clock time, the answer to the same query may change in the future, since application states may change due to the recovery. Answers to the queries for the time older than $T_{current} - T_{recovery}$ are consistent although they are not necessarily accurate.

The worst case size of delay and recovery buffer in terms of number of event elements can be calculated based on the maximum event rate and the time length of two buffers:

$$Size_{delay} = T_{delay} \times Max_{eventrate}$$
$$Size_{recovery} = (T_{recovery} - T_{delay}) \times Max_{eventrate}$$

[1]Recall that the temporal evolution of the state is preserved by the system to answer time-series queries ("Give the track of person A from 10 AM to 5 PM").

VI. Evaluation

To evaluate our prototype framework with a complete application, we used existing algorithms. For signature generation, we used Viola-Jones face detection algorithm [9] from the OpenCV [26] library in the *detect_signature* handler. The face detection algorithm finds all faces from a single video frame, and reports the bounding boxes of the faces.

For event generation, we use face recognition algorithms from OpenCV. Currently there are three algorithms that are used in the *generate_event* handler: Eigenface, Fisherface, and LBPH. Each algorithm has different computational complexities and accuracies. However, we do not consider algorithmic accuracies since we focus on solving performance bottlenecks for large-scale spatio-temporal analysis. When the algorithms are invoked, distances between a detected signature and known signatures are calculated. Once the distances are calculated, we transform the distance values to normalized similarities (i.e., probabilities for each known signature) based on the method developed by Bouchaffra et al. [24]

After an event is generated, the event is used for state update. Specifically, we implemented the state update algorithm proposed by Menon et al. [27] within the *update_state* handler. In the algorithm, each state element represents a probability of an occupant being in a specific zone. For the given event element $p(o_i)$ tagged with an occupant ID o_i and zone ID z_i, the new probability for an occupant, $p_s(o_i)$ for the zone z_i is calculated as follows: $p_s(o_i) = p(o_i) + x_i * p'_s(o_i)$, where $x_i = 1 - p(o_i)$ and $p'_s(o_i)$ is the probability of the occupant in the previous state. For all other zones, $p_s(o_i) = x_i * p'_s(o_i)$. This ensures that the sum of probabilities for an occupant across all zones in the new state transition table equals 1.

We evaluate our framework with various resource configurations using Amazon EC2 as an elastic computing infrastructure. Following parameters specify different configurations for our experiments:

$N_{ocupant}$	Number of known occupants in the system
N_{zones}	Number of zones in the system
$N_{workers}$	Number of workers
$Rate_{event}$	Event rate
$Rate_{signature}$	Signature rate
$Nodetype$	Instance type in Amazon EC2
$Algorithm$	Algorithm name

The following sections discuss the various performance aspects of large-scale spatio-temporal analysis using our framework.

A. Workload for Spatio-temporal Analaysis

In this section, we evaluate the performance for each step of spatio-temporal analysis using the design choice made in the Section III. For signature detection, we used a local desktop to mimic an embedded smart camera, while Amazone EC2 is used as an elastic computing infrastructure.

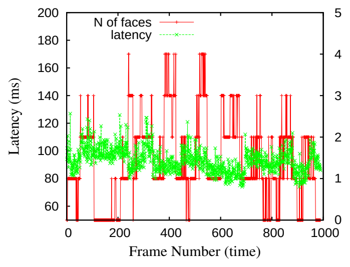
Figure 10(a) shows the computing latency of face detection for each video frame on a local desktop. We used a 640 x 480 video captured from a surveillance camera, and performed a Viola-Jones face detection algorithm. While measuring the latency for each frame, we also recorded the number of faces detected from each frame. As shown in the figure, the latency for face detection varies mostly from 80 to 120 milliseconds, regardless of the number of faces detected. This is because the face detection algorithm searches for faces in all regions of an image, and therfore its cost depends on the size of raw video frame, rather than the number of faces. Such a stable computational workload of face detection algorithms makes it possible to perform using local resources in an embedded smart camera.

Figure 10(b) shows the computing latency of a face recognition algorithm, Eigenface [12], on differrent types of computing resources in Amazon EC2. In this experiment, different face images from LFW dataset [28] are used as input signatures. As shown in the figure, each EC2 instance type has different amount of system resources and therefore the average latencies for face recognition are different depending on the instance type. Plus, all instance types except m3.xlarge show fluctuations in their computing latencies. We think this is due to the virtual machines that share physical CPU time, while the high-end instance type has a dedicated physical CPU. However, all instance types show stable average latencies in the long term, which allows predictive resource provisioning based on the off-line profiling.

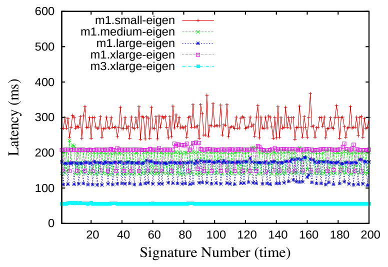
For state update, we evaluate the network latencies rather than the computing latencies since computing latencies on elastic computing resources are already presented in Figure 10(b). Figure 10(c) shows the cumulative distribution function of network latency for the phase 2 of state update for processing one event per second. In this experiment, we measured network latency of each message transfer between every pair of state workers, while no selective heuristics are used; all occupant and zone states are updated and therefore all state workers will transmit a message to each other for a single event. For different setup, we used different number of state workers but the number of occupants and zones remain the same. This means the more the workers, the smaller the message size for each pair of state workers since each state worker is responsible for smaller number of occupant and zone states. The size of each message and the total number of message exchanges among state workers, (not counting local communication within a single state worker) is as follows:

$$MsgSize = \frac{N_{occupants} \times N_{zones}}{(N_{workers})^2}$$
$$TotalTransfer = (N_{workers} - 1) \times N_{workers}$$

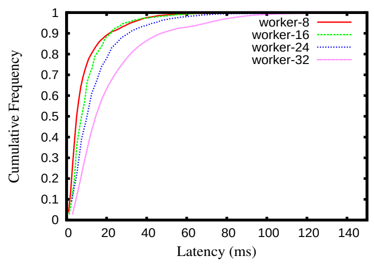
As shown in Figure 10(c), a setup with more state workers shows higher average network latency for each message, with more variance in latencies. This is because the total



(a)



(b)



(c)

Figure 10. Performance for each step in spatio-temporal analysis is measured: (a) Computing latency for signature generation: OpenCV face detection algorithm is performed at a local desktop with Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz, using a 640 x 480 video stream. (b) Computing latency for event generation: $Algorithm$=Eigenface, $N_{occupants}$=425 (c) Cumulative distribution function of communication latencies for state update: $N_{occupants}$=425, N_{zones}=1000, $Nodetype$=m1.medium

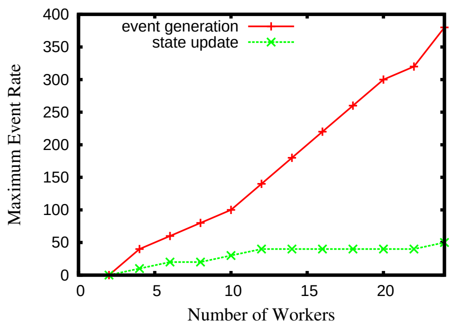


Figure 11. Maximum event rates with different number of nodes for event generation and state update: $N_{occupants}$=425, N_{zones}=1000, $Algorithm$=LBPH, $Nodetype$=m1.medium

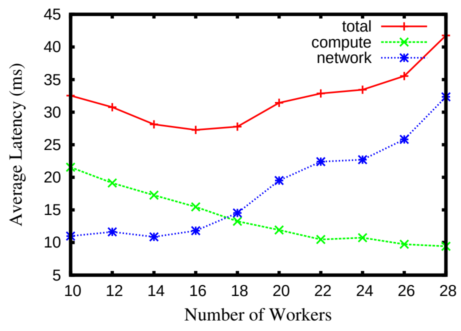


Figure 12. Average computation and network latency for state update in underloaded condition: $Rate_{event}$=10, $N_{occupants}$=425, N_{zones}=1000, $Nodetype$=m1.medium

number of messages increases exponentially with the number of state workers used, although the total amount of data exchanged among state workers increases only slightly.

B. Scalability of Event Generation and State Update

To evaluate scalability of spatio-temporal analysis, we performed stress test on event generation and state update. Specifically, we increased event rate on a specific resource configuration, until the latencies saturate; the system is overloaded from that point. We define the maximum event rate for the specific configuration as the event rate right before the latency saturation.

Figure 11 shows the maximum event rate of event generation and state update. Event generation scales well since events are generated on different workers, and putting more event workers will linearly increase the maximum event rate. However, state update without selective heuristics has a poor scalability since the maximum event rate does not increase with more workers.

To investigate the reason for the poor scalability, we measured the detailed cost for state update. Figure 12 shows the total latency, computation latency and network latencies of state update for different number of state workers. The event rate (10 events per second) is carefully selected so that all configurations are not overloaded for the event rate. As the graph shows, computing latency decreases when more worker nodes are used. However, the network latency starts to increase from 16 worker nodes , which makes the total latency also increase from 20 worker nodes. This shows that the scalability of state update is bounded because the network latency starts to dominate from a certain number of state workers.

C. Effect of Selective State Update Heuristics

In large-scale spatio-temporal analysis, it is necessary to use selective heuristics for state update since the naive state update causes poor scalability. In this section, we show the impact of the selective heuristics on system performance.

Figure 13 shows different average latencies of various heuristics with different selectivity, while varying the number of occupants in the system. The heuristics use *set_occupant_mask()* in the *generate_event* handler to mask all event elements that are not selected. For example, select-1 scheme will select only a single event element while all other event elements are masked. As shown in the figure, the naive state update selecting all event elements (select-

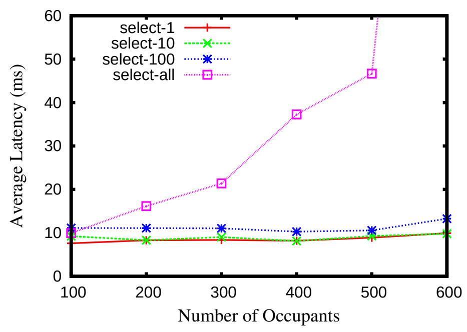


Figure 13. Average latency of state update for different number of occupants selected: N_{node}=8 , $Rate_{event}$=40, $Nodetype$=m1.medium, N_{zones}=1000

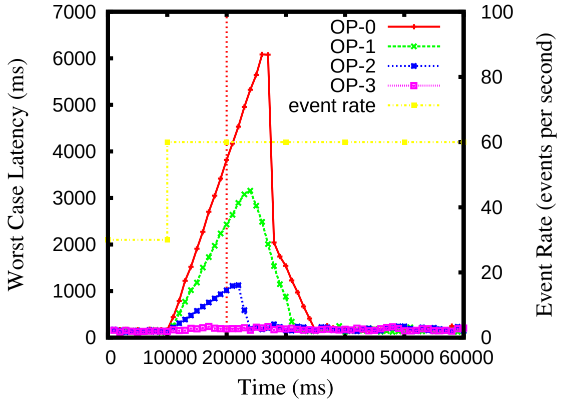


Figure 14. Quality of service of event generation during the dynamic resource adaptation: $Rate_{event}$=30 to 60, $N_{occupants}$=425, $Algorithm$=LBPH

all) scales poorly, and the system is overloaded when more than 500 occupants are in the system. Other heuristics show good scalability, where the average latency depends on the number of occupants selected, and not on the total number of occupants in the system.

D. Impact of Overprovisioning on the Quality of Service

While an application is running, our framework adapts to the current workload by changing the resource configuration. To handle an increased workload, the framework starts provisioning more resources based on the estimated current workload. After the resource provision delay that depends on the underlying elastic computing infrastructure, resources become available and are used to handle the increased workload. Figure 14 shows the quality of service for event generation during such an adaptation. In this controlled experiment, we measured the worst case latency for each time period (1 second) as a measure of quality of service. Different overprovisioning polices are used with different amount of overprovisioned resources. For example, OP-0 keeps the number of workers that are just enough to handle the given workload, while OP-2 keeps two more workers than necessary. For simplicity, we assume that the workload change is immediately captured by the resource manager in our framework. However, we assume that there is a 10 second delay for resource provisioning.

At the beginning of the experimental scenario, the event rate is 30 events per second, and there are 4 event workers handling the workload. The quality of service is around 150 milliseconds for all different policies, since the system is not overloaded. After 10 seconds, the workload is suddenly increased to 60 events per second, and the resource manager starts additional resource provisioning immediately. As can be seen from the figure, the more over-provisioning there is the more graceful is the handling of the increased workload during the time when the resource manager is trying to allocate new resources. After 10 seconds, the newly provisioned resources become available and the system has fully recovered and adjusted to the increased workload.

VII. Conclusion

Distributed smart camera networks are being widely deployed from building scale to city scale. Spatio-temporal analysis is one of the key techniques, converting raw video streams to knowledge of occupants' whereabout to enable a wide range of applications such as surveillance, transportation, assisted living, and the like. However, there are serious impediments to scaling such techniques to large-scale camera networks, as it involves developing the right programming middleware and resource management infrastructure for handling the dynamic workload in real time.

In this paper, we tackle the technical challenges for developing a real-time spatio-temporal analysis application on a large-scale camera network. We present a novel distributed framework that minimize the burden on the domain experts by just requiring them to provide the domain-specific handlers and quality of service parameters. Our system provides automatic resource configuration through profiling each step of the spatio-temporal analysis with given application-specific handlers on the elastic computing resources. We have implemented our system using Amazon EC2 and have conducted detailed performance evaluations showing the scalability of our framework.

VIII. Acknowledgment

Our sincere thanks to Amazon for generously providing the computational resources on EC2 for carrying out our experiments. Our thanks are due to our colleagues in the embedded pervasive lab, particularly Dave Lillethun.

References

[1] "Schiphol Airport leverages technology to drive efficiency." [Online]. Available: "http://www.securitysystemsnewseurope.com/?p=article&id=se200906VF2Isw"

[2] U. Ramachandran, K. Hong, L. Iftode, R. Jain, R. Kumar, K. Rothermel, J. Shin, and R. Sivakumar, "Large-scale situation awareness with camera networks and multimodal sensing," *Proceedings of the IEEE*, vol. 100, no. 4, pp. 878 –892, april 2012.

[3] V. Menon, B. Jayaraman, and V. Govindaraju, "The Three Rs of Cyberphysical Spaces," *Computer*, vol. 44, pp. 73–79, 2011.

[4] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer, "EasyLiving: Technologies for Intelligent Environments," in *Handheld and Ubiquitous Computing*, ser. Lecture Notes in Computer Science, P. Thomas and H.-W. Gellersen, Eds. Springer Berlin / Heidelberg, 2000, vol. 1927, pp. 97–119.

[5] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *Pervasive Computing, IEEE*, vol. 1, no. 4, pp. 74 – 83, oct-dec 2002.

[6] P. S. Pillai, L. B. Mummert, S. W. Schlosser, R. Sukthankar, and C. J. Helfrich, "Slipstream: scalable low-latency interactive perception on streaming data," in *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video*, ser. NOSSDAV '09. New York, NY, USA: ACM, 2009, pp. 43–48.

[7] R. Feris, A. Hampapur, Y. Zhai, R. Bobbitt, L. Brown, D. Vaquero, Y. Tian, H. Liu, and M.-T. Sun, "Case-Study: IBM smart surveillance system," in *Intelligent Video Surveillance: Systems and Technologies*, Y. Ma and G. Qian, Eds. Taylor & Francis, CRC Press, 2009.

[8] K. Hong, S. Smaldone, J. Shin, D. J. Lillethun, L. Iftode, and U. Ramachandran, "Target container: A target-centric parallel programming abstraction for video-based surveillance," in *ICDSC*, 2011, pp. 1–8.

[9] P. Viola and M. Jones, "Robust real-time face detection," *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004.

[10] R. Hsu, M. Abdel-Mottaleb, and A. Jain, "Face detection in color images," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 5, pp. 696–706, 2002.

[11] E. Hjelmås and B. Low, "Face detection: A survey," *Computer vision and image understanding*, vol. 83, no. 3, pp. 236–274, 2001.

[12] M. Turk and A. Pentland, "Eigenfaces for recognition," *Journal of cognitive neuroscience*, vol. 3, no. 1, pp. 71–86, 1991.

[13] L. Wiskott, J. Fellous, N. Kuiger, and C. von der Malsburg, "Face recognition by elastic bunch graph matching," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 19, no. 7, pp. 775–779, 1997.

[14] V. Blanz and T. Vetter, "Face recognition based on fitting a 3d morphable model," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 9, pp. 1063–1074, 2003.

[15] B. Moghaddam, T. Jebara, and A. Pentland, "Bayesian face recognition," *Pattern Recognition*, vol. 33, no. 11, pp. 1771–1782, 2000.

[16] A. Nefian and M. Hayes III, "Hidden markov models for face recognition," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 5. IEEE, 1998, pp. 2721–2724.

[17] L. Wang, T. Tan, H. Ning, and W. Hu, "Silhouette analysis-based gait recognition for human identification," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 12, pp. 1505–1518, 2003.

[18] A. Kale, N. Cuntoor, B. Yegnanarayana, A. Rajagopalan, and R. Chellappa, "Gait analysis for human identification," in *Audio-and Video-Based Biometric Person Authentication*. Springer, 2003, pp. 1058–1058.

[19] R. Frischholz and U. Dieckmann, "Biold: a multimodal biometric identification system," *Computer*, vol. 33, no. 2, pp. 64 –68, feb 2000.

[20] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, "Face recognition: A literature survey," *ACM Comput. Surv.*, vol. 35, no. 4, pp. 399–458, 2003.

[21] C. V. Design, *Cisco IP Video Surveillance Design Guide*. Cisco, 2009.

[22] A. Elgammal, R. Duraiswami, D. Harwood, and L. S. Davis, "Background and foreground modeling using non-parametric kernel density estimation for visual surveillance," *Proceedings of the IEEE*, July 2002.

[23] T. Parag, A. Elgammal, and A. Mittal, "A framework for feature selection for background subtraction," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2006.

[24] D. Bouchaffra and V. Govindaraju, "A methodology for mapping scores to probabilities," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 21, no. 9, pp. 923 –927, sep 1999.

[25] D. Mills, "Network time protocol (version 3) specification, implementation and analysis," 1992.

[26] "OpenCV Wiki," 2010. [Online]. Available: http://opencv.willowgarage.com/wiki/

[27] V. Menon, B. Jayaraman, and V. Govindaraju, "Multimodal identification and tracking in smart environments," *Personal Ubiquitous Comput.*, vol. 14, pp. 685–694, December 2010. [Online]. Available: http://dx.doi.org/10.1007/s00779-010-0288-6

[28] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments," University of Massachusetts, Amherst, Tech. Rep. 07-49, October 2007.