ThingStore: A Platform for Internet-of-Things Application Development and Deployment

Kutalmış Akpınar
Dept. of Electrical Engineering
& Computer Science
University of Central Florida
Orlando, FL 32816, USA
kutalmis@knights.ucf.edu

Kien A. Hua
Dept. of Electrical Engineering
& Computer Science
University of Central Florida
Orlando, FL 32816, USA
kienhua@eecs.ucf.edu

Kai Li
Dept. of Electrical Engineering
& Computer Science
University of Central Florida
Orlando, FL 32816, USA
kaili@eecs.ucf.edu

ABSTRACT

An advanced app-store concept, called *ThingStore*, is introduced in this paper. It provides a "market place" environment to facilitate collaboration on Internet-of-Things (IoT) applications development, and a platform to host their deployment. *ThingStore* services three categories of users: (1) Thing Provider - "Things" (such as online cameras and sensors) can be made more intelligent through event detection software routines called smart services. A thing provider may deploy "things" and advertise their smart services at *ThingStore* market place. (2) Software Developer - Software developers can develop apps that query relevant smart services using EQL (Event Query Language) much like the way traditional database applications are conveniently developed atop a standard database management system today. (3) End User - An end user may subscribe to a particular app for event notification and management. In this IoT architecture, *ThingStore* is a computation hub that links together human,"things," and computer software in a cyber-physical lifecycle to enable fusion of human and machine intelligence to accomplish some common goal. Not only human, but also "things," may adjust the physical world. New changes in the physical world may, in turn, incur new event detections and therefore initiate another cycle of this ecology-inspired computational lifecycle.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous;
D.2.11 [**Software Engineering**]: Software Architectures

Keywords

ThingStore; Internet of Things; data stream processing; complex event processing; event query language; service-oriented architecture.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'15, June 29 - July 3, 2015, OSLO, Norway.
Copyright 2015 ACM 978-1-4503-3286-6/15/06...$15.00.
http://dx.doi.org/10.1145/2675743.2771833

1. INTRODUCTION

The Internet has evolved to include not only computers, but also devices of all types. The Internet of Things (IoT) is the interconnection of uniquely identifiable devices with associated computing services which can be performed on embedded computing devices or on an external computing server. In the IoT environment, physical objects (i.e., "things") such as cameras, sensors, and actuators, among others are equipped with the capability to communicate over the Internet without human intervention. They form a cyber-physical system that enables many advanced applications [21, 28, 16]. There are several definitions for the IoT. They all share the fact that there are three types of communication networks, namely the autonomous field networks, the wide area networks, and the Internet. Examples of field networks are Radio-Frequency Identification (RFID) networks, wireless sensor networks, and vehicular networks. Other field networks include WiFi, Bluetooth, and ZigBee, where devices can be tablets, laptops, mobile phones, and home appliances, etc. These autonomous field networks are interconnected via long range wired or wireless access network that facilitates access to online information through the Internet. In this paper, we focus on the computational aspect of IoT and introduce *ThingStore*, a generic platform for IoT application development and deployment.

Due to a lack of standards on the architecture of the IoT, the status quo is that there is an intranet of things with many "islands" of field networks existing in silos, each supporting different IoT applications. This phenomenon is also due to the heterogeneity of devices; and addressing their interoperability remains a great challenge. We need a unified architecture in order to seamlessly integrate these decentralized environments with different types of "things" for Internet-scale applications.

The design of a unified architecture poses several challenges when dealing with vast amount of information continuously flowing from "things." First, maintaining connectivity and achieving realtime processing of so many data streams is not feasible for many application developers with limited budgets; second, the heterogeneity of "things" requires software developers to have a wide variety of expertise to deal with all kinds of data (e.g. medical imaging, physics, etc.), which is overly demanding in practice; and finally, network bandwidth and computation are wasted if data from "things" are continuously transferred over the network to the applications for computation. Those limitations call for IoT architectures that can ease software development and

deployment and improve the computation efficiency of the IoT systems.

We introduce *ThingStore* in this paper. It is designed to address the aforementioned challenges and achieve the desired functionality in the following ways:

- Dealing with device heterogeneity: "Things" are made intelligent with event detection software routines, called smart services, as interface. These smart services carry out computations upon raw data generated by their physically collocated or decoupled sensing components to detect specific events. These smart services provide a binary abstraction ("1" if an event is detected and "0" otherwise) to hide the complexity due to heterogeneity of "things" (e.g., cameras vs environmental sensors).

- Dealing with system complexity: Applications can be developed atop the smart services through a unified binary interface using EQL (Event Query Language). This framework enables collaboration between the developers of the smart services and the application developers. The latter can focus on the application logic and do not have to be expert on thing-specific computation such as computer vision techniques. The decouple of the thing-specific computation from the application logic also avoids otherwise duplicated computation in many IoT applications.

- Dealing with efficient use of network bandwidth: The smart services can be pushed closer to the live data sources (i.e., "things") to avoid non-stop streaming of sensor and video data over the network to the applications. This is particularly important for Internet-scale deployment. On-demand video streaming already accounts for more than 70% of Internet traffic today. Non-stop IoT streaming would add substantially more traffic to this percentage. A recent study by the business consultancy firm Gartner anticipates 26 billion Internet-connected "things" by 2020.

- Dealing with expensive system costs: *ThingStore* is an advanced app-store concept. It provides a "market place" environment to facilitate collaboration on IoT applications development, and a platform to host their deployment. A thing provider may deploy "things" and advertise their smart services at *ThingStore* market place. Application developers can develop apps that query relevant smart services using EQL (Event Query Language) much like the way traditional database applications are conveniently developed atop a standard database management system today. End Users may subscribe to various apps for different IoT services. With this architecture, the *ThingStore* ecosystem allows cost sharing of large-scale IoT applications development and deployment.

The remainder of this paper is organized as follows. We introduce the *ThingStore* environment for IoT in Section 2. System design is presented in Section 3. We describe a system prototype and provide some experimental results in Section 4. Related work is discussed in Section 5. Finally, we conclude this paper and discuss our future work in Section 6.

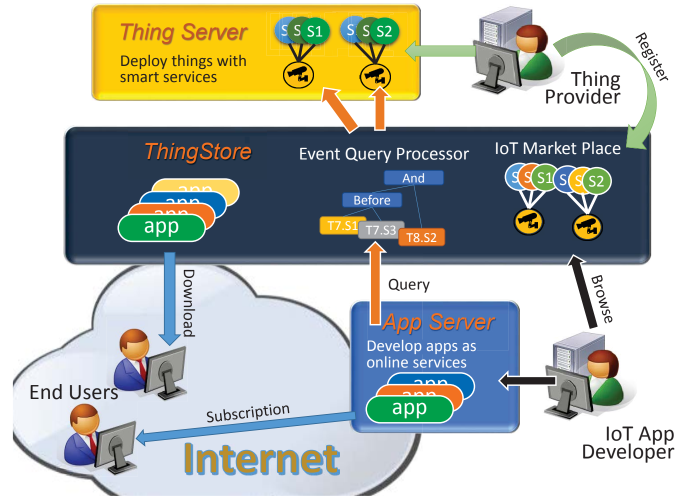

Figure 1: The IoT environment and ThingStore

2. THINGSTORE ENVIRONMENT

As an IoT platform, *ThingStore* brings together three different actors of cyber-physical environment: thing provider, software developer and end user. The overall architecture of *ThingStore* is illusated in Figure 1 and several deployment options based on it are discussed in this section.

Thing providers can be enterprises, government institutions or individuals who aim to deploy "things" to reach a potential market. Traditionally, "things" are deployed to serve specific applications. In *ThingStore*, however, "things" are treated as infrastructural assets that are better utilized when shared by applications. For instance, a common set of traffic cameras may be simultaneously utilized by applications that serve the highway agency (e.g. for traffic rule enforcement), the police department (e.g. for security monitoring), or the research facility (e.g. automatic traffic video analysis). In addition to making "things" more useful, "things" sharing reduces costs for software developers with limited infrastructural budget and makes it possible for "thing" providers to gain more profits.

As shown in Figure 1, *ThingStore* life cycle starts with the deployment of "things", whose sensing components are the primitive sources of data to be consumed by applications. "Things" become "intelligent" when bundled with smart services, which are software routines that process and transform raw data into meaningful information. For example, a smart camera may be paired with a computer vision-based anomaly detection service that triggers notifications upon the detection of an abnormal event. Simple services that require little computation may run on the embedded computing component; while more sophisticated ones may be hosted from the physically-decoupled "thing" servers deployed by the "thing" provider. *ThingStore* also provides a market place where the "thing" providers can advertise their "things" and services. This *ThingStore* approach makes "thing" providers' infrastructure more valuable and desirable for their customers.

In order to deal with device heterogeneity, *ThingStore* exposes a service definition library for the "thing" providers to define smart services and interface with *ThingStore*. The

standard interface definitions form the basis of event hosting, event subscription and event query.

Software developers can develop applications atop *ThingStore* and query interested events much like the way database applications interact with a standard database management system today. *ThingStore* provides an SQL-like query language for applications to define and query events and a web-service interface for the ease of connectivity. When "things" seamlessly scale in time, the same application will continue to benefit from the environment without additional requirements or software revision. Since event computation and connectivity with huge amount of "things" are handled by the "thing" providers and *ThingStore*, software developers only have to deal with lightweight development on mobile devices or personal computers. Morever, the IoT applications developed atop *ThingStore* can be hosted from application servers with lower costs. And this brings about twofold benefits: first, it provides opportunities for open innovation and helps more software developers to contribute to the platform; second, large enterprises who want to run their own analytics applications on their own servers can also query *ThingStore* for their own interests, and serve additional applications to the users.

End users can interact with the IoT environment through the applications which provide GUI and abstraction for their specific needs. In response to the occurrence of an event, queries may provide notification or lead to interactions with other "things", thus enabling collaboration between them. In a broader sense, *ThingStore* acts as a platform to bring thing providers and application developers together for the convenience of serving end users.

3. THINGSTORE DESIGN

ThingStore is the core of the proposed IoT environment. The system components of *ThingStore* are illustrated in Figure 2. It performs the following four primary tasks:

1. *ThingStore* provides service definition APIs for smart service development and hosting. "Things" host their smart services on Thing Servers and enable *ThingStore* to make connections to these smart services via the aforementioned API.
2. *ThingStore* maintains the meta information about "things" and their smart services. This is achieved through the market place, a relational database with *StoreBuilder* as the administration interface and the thing browsing service as the marketing environment.
3. *ThingStore* hosts event queries written in EQL (Event Query Language) and manages user subscriptions to these queries for the event notification service. *ThingStore* also supports programming interface for applications to submit and execute EQL queries at run time. Alternatively, applications can also call pre-compiled EQL queries at *ThingStore*. This is similar to stored procedures in SQL environment enhanced with novel mechansm to allow applications to interact with continuous queries.
4. *ThingStore* includes a query processor to support real-time event query processing. Both inter-query parallelism and intra-query parallelism are supported.

We describe the aforementioned functionality of *ThingStore* in more detail in the remainder of this section.

3.1 IoT Market Place

The IoT market place maintains a relational database for thing providers to submit things and their services. An administrator interface, *StoreBuilder*, is created for managing the repository of things and services. Data in market place can be accessed by *StoreBuilder*, IoT applications, and the event query processor with different permission levels. *StoreBuilder* has the highest access permissions for updating database elements. While application developers and their applications only have permissions to read meta information such as thing/service descriptions, thing locations, and providers, etc. Event query processor has access to the market place in order to retrieve metadata of services such IP addresses, access path, and communication protocols etc.

For uninterrupted service provisioning, *StoreBuilder* operations can be concurrently performed on *ThingStore* while the event queries are being processed. This feature is provided by setting appropriate delete cascades in the database, so that *StoreBuilder* operation does not leave the database in an inconsistent state.

3.2 Event Data Modeling

ThingStore uses event query processing to handle complex event detection. Event queries are continuous queries computed in real-time on live data streams. The proposed query processing model abstracts away heterogeneity of "things" and provides mechanisms for the composition of arbitrarily complex events. Such sophistication builds upon a generic and unified event data model. Details of those architectural components shall be presented in the next few sections.

DEFINITION 1. *A* ***data frame*** f_t *is the raw data generated by a thing at the discrete time point* t.

DEFINITION 2. *A* ***smart service*** S *is a computer program that computes on an ordered sequence of data frames and produces boolean results. Formally, it is defined as:*

$$S : \mathbf{f}_L \rightarrow \{0, 1\},$$

where $\mathbf{f}_L = (f_1, f_2, \cdots, f_L)$ *is a sequence of data frames in ascending temporal order,* L *is the size of the computation window and '0' and '1' represent false and true respectively.*

Different types of "things" in the internet generate different kinds of data. For example, temperature sensors generate numeric readings; video cameras produce image frames and/or audio signals; GPS devices gather coordinates information. With the smart service modeling approach, such diverse data of heterogeneous "things" are transformed into a universal boolean representation encoding the processing results of pre-defined events.

DEFINITION 3. *A* ***dynamic relation*** *is an in-memory FIFO "table" where each row has a discrete time stamp and each column is associated with some computer program. An entry in this "table" is a boolean value indicating the computation result of the corresponding computer program at a specific discrete time point.*

DEFINITION 4. *An* ***incident*** e_t *is a '1' entry in the dynamic relation representing a "true " result returned by the computer program at time* t.

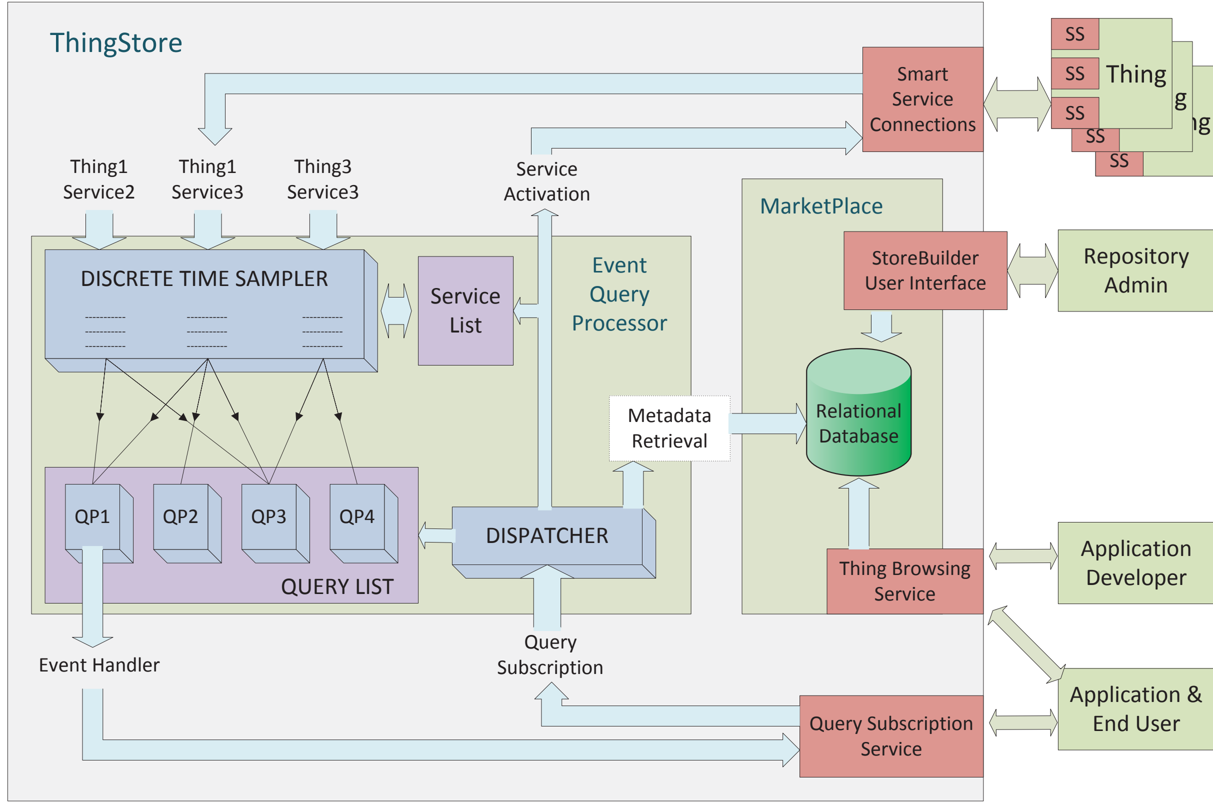

Figure 2: The system components of *ThingStore*.

DEFINITION 5. *An* ***event*** E *is a continuous sequence of incidents with a length greater than or equal to a predefined threshold* ΔT_{min}*. Formally, it's defined as*

$$E : (e_{t_s}, e_{t_s+1}, \cdots, e_{t_e}),$$

where t_s *and* t_e *are the starting and ending time points of the event, respectively; and* $t_e - t_s \geq \Delta T_{min}$*. An event can be compactly represented as a interval* (t_s, t_e)*.*

DEFINITION 6. *An* ***atomic event*** *is an event whose incidents are produced by a smart service.*

By definition, a smart service is essentially an event detector for atomic events, therefore we do not strictly differentiate those two terms and use them interchangeably in the rest of the paper.

DEFINITION 7. *A* ***base dynamic relation*** *is a dynamic relation for a particular "thing" where the computer programs associated with the columns are smart services registered with this "thing". It can be defined according to the schema*

$$R_d(S_1, S_2, \cdots, S_n),$$

where "S_i*;* $i = 1; 2; \cdots ; n$*" are active smart services of this "thing." The number of rows of a base dynamic relation is the maximum query window size of the active queries associated with this dynamic relation.*

The dynamic relation as illustrated in Figure 3 is the basic data structure in *ThingStore* upon which complex event

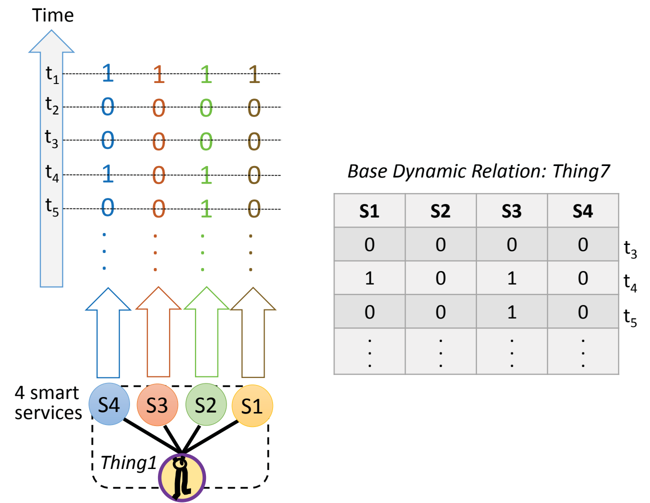

Figure 3: Base dynamic relation of a "thing".

queries are formulated. Conceptually, it's similar to the table in a traditional relational DBMS. However, there are several major differences: first, stored data in dynamic relations are indicators of computation results on thing's data rather than the raw data itself; second, a dynamic relation is continuously updated as it only stores a snapshot of the most up-to-date window of computation results; and third,

relational algebra on traditional relations does not apply to dynamic relations as the two types of tables have different data semantics.

Modeling "things" as dynamic relations abstracts away the underlying heterogeneity of internet "things," and allows us to build a set of operations upon this common data structure.

Definition 8. *A **logical operator** op_l evaluates the logical relationship between two events. It's defined by the mapping*

$$op_l : (c_{1,t}, c_{2,t}) \to \{0, 1\},$$

where $c_{1,t}$ and $c_{2,t}$ are two entries (i.e., two boolean values) at the same discrete time point t in one or two dynamic relations; and op_l can be any element in the following set:

$$LOP = \{And, Or, Not\}.$$

Definitions of the elements in LOP are the same as the basic logical operators in Boolean algebra.

Definition 9. *A **temporal operator** op_t evaluates the temporal relationship between two events. It's defined by the mapping*

$$op_t : (\mathbf{c_1}, \mathbf{c_2}) \to \{0, 1\}$$

where $\mathbf{c_1}$ and $\mathbf{c_1}$ are columns (i.e., vectors of boolean values) of one or two dynamic relations, and op_t is an element of the set:

$$TOP = \{Before, Meets, During, Overlaps, Starts, Ends\}.$$

Let (t_{s1}, t_{e1}) and (t_{s2}, t_{e2}) be the interval representation of two events E1 and E2, elements in TOP are defined as:

$$\begin{aligned}
Before(E1, E2) &\iff t_{e1} > t_{s2} \\
Meets(E1, E2) &\iff t_{e1} = t_{s2} \\
During(E1, E2) &\iff t_{s1} < t_{s2} \&\& t_{e1} > t_{e2} \\
Overlaps(E1, E2) &\iff t_{s1} < t_{s2} < t_{e1} < t_{e2} \\
Starts(E1, E2) &\iff t_{s1} = t_{s2} \&\& t_{e1} < t_{e2} \\
Ends(E1, E2) &\iff t_{s1} > t_{s2} \&\& t_{e1} = t_{e2}
\end{aligned}$$

Our definition of temporal operators follows the classic definition by Allen et al.[3]. These simple constructs allow formulation of queries on arbitrarily complex spatiotemporal events that may involve many dynamic relations corresponding to different Internet "things." An example of a query that involves two "things," two temporal operators, and one logical operator is illustrated in Figure 4. Since each of the two "things" has four smart services, the two corresponding base dynamic relations each have four columns (Tables *Thing1* and *Thing7*). The expression tree in Figure 4 represents the execution plan for the aforementioned continuous query. As the two base dynamic relations are refreshed with new data entries computed by their smart services at each time step, each of the three operators "Before," "And," and "Meet" computes a "0" or a "1" as output accordingly. We can create another dynamic relation with three columns, each storing the outputs computed by one of the three spatiotemporal operators (see the top table in Figure 4). This table is refered to as the *dynamic view* of this query. Its entries capture detection of complex incidents defined in terms of the atomic incidents captured in the base dynamic relations. In particular, a continuous sequence of "1" etnries in the column corresponding to the root of the expression tree (i.e., the "And" column) indicates a detection of the query event if this "1" sequence is longer than a predefined threshold. This is referred to as a *composite event*.

Definition 10. *A **dynamic view** of a query is a dynamic relation where the data entries in each column is computed by a distinct spatiotemporal operator used in the query. It can be defined by the following schema*

$$V_d(op_1, \cdots, op_i, \cdots),$$

where $op_i \in LOP \bigcup TOP$ and is used in the query. They are the computational operator nodes in the query execution plan.

Definition 11. *A **composite event** is an event whose incidents are from the dynamic view.*

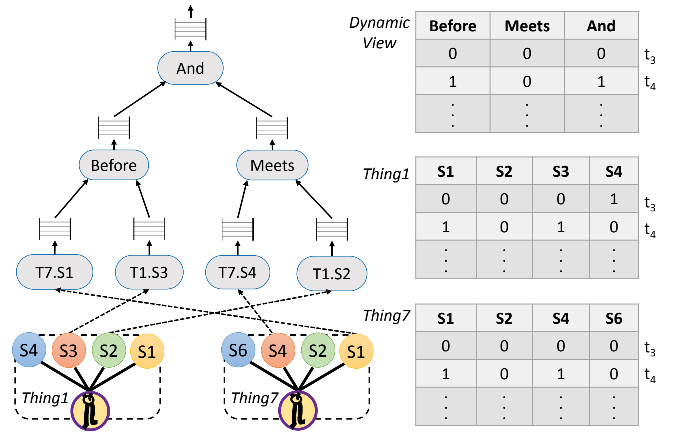

Figure 4: Left: expression tree of the event "Before(T7.S1, T1.S3) And Meets(T7.S4, T1.S2)". Right: dynamic views and base dynamic base relations involved in this event.

3.3 Smart Service API

ThingStore communicates with smart services through a standard interface. Definition of the interface is provided as a class library. In order for seamless integration with the *ThingStore* platform, smart services are defined by inheriting the interface of this class library.

Smart services are typically hosted by the physically coupled/decoupled computational component of "things" and allow for access from *ThingStore*. It holds the service address information in a relational database and selects what services to connect at runtime. This saves communication overhead of keeping every smart-service connection alive. Hosting smart services separately, rather then on *ThingStore*, gives flexibility to the deployment of "things." An address change of *ThingStore* does not require configuration changes in every smart service, and smart services will have less data to hold. Updates on service addresses can be easily handled through *StoreBuilder*. Moreover, the same interface can be optionally connected by clients other than *ThingStore*.

Smart service interface defines a list of actions requested by *ThingStore* and a callback to *ThingStore* (Figure 5). The list of actions includes *activate*, *beginstream*, and *stopstream*.

The callback is defined as a single function, *eventstate*, which accepts a boolean atomic event as its parameter. As shown in Figure 5, communication between *ThingStore* and the smart service starts by an *activate* call to the service. Depending on the implementation of the service provider, the service can be configured to run constantly, or it can wait for this call to start running. Return from the service call is an acknowledgment of event computation. When *ThingStore* is ready for accepting the event states (a bit stream), it calls *beginstream* to receive the atomic events. This call immediately returns, and it signals the smart service to stream the event states. In its event streaming, by calling the callback function *incidentstate*, the smart service reports a boolean event state after every computation cycle of event detection routine, or in every state change (changing from "0" to "1", or vice versa). An event stream can be terminated by calling *stopstream*; and the deactivation of event computation can be triggered from *ThingStore*'s disconnection.

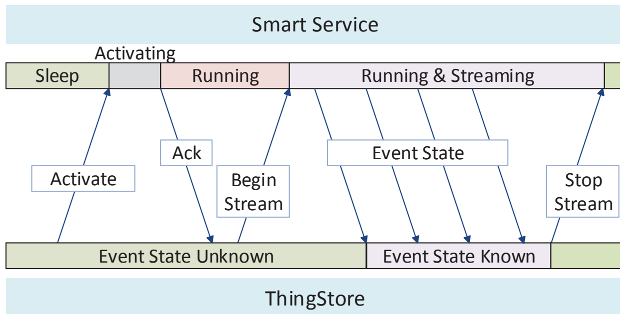

Figure 5: Communication protocol between a smart service and *ThingStore*

3.4 Event Query Language

In *ThingStore*, event queries are formulated using Event Query Language (EQL). The basic syntax of an EQL query follows a SQL-like SELECT-FROM-WHEN-WITHIN-UNTIL statement. The language features can be best described in terms of the following event query example:

SELECT Event-handler
FROM Thing7 AS T7, Thing9 AS T9
WHEN Before(T7.S1, T9.S3)
WITHIN 30 seconds
UNTIL 5 minutes

This event query is based on two smart services from two smart "things" (*Thing7* and *Thing9*). In this EQL query, the event-handler is the computer program to be activated should the event specified in this query become true. The FROM clause shows the relevant dynamic relations Thing7 and Thing9 with their alias names T7 and T9, respectively.

The composite event is specified in the WHEN clause as an interval-temporal-logic expression on two atomic events. It describes the composite event as "The smart service S1 at thing T7 detects an atomic event before the smart service S3 at thing T9 detects an atomic event." As defined in the previous section, EQL provides a rich set of temporal and logical operators to allow expression of very complex events.

The WITHIN clause signifies the event window (time duration for event detection). That is, the event specified in the WHEN clause must occur within a 30-second period. Since the query processor needs only remember the incidents within the last 30 seconds in order to answer this query, the dynamic relations T7 and T9 need only retain incident detection results from the last 30 seconds. These data tables are therefore typically very small and can be kept in computer memory to leverage the efficiency of in-memory query processing.

The UNTIL clause in the above EQL query indicates the duration of this query. It specifies that the execution of this continuous query terminates with a "true" (i.e., "1") result if an event is detected within 5 minutes; otherwise, it automatically terminates at the end of the fifth minute.

ThingStore supports both stored queries and API for submitting queries from applications. Once a stored query is registered with *ThingStore*, end users may subscribe to this query for event notification. Applications may also subscribe to a stored query by calling the query from the host language similar to calling stored procedure in SQL. The API call interface is similar to JDBC. The behavior is like calling a binary procedure with the result being a continuous sequence of "true" or "false". The UNTIL clause is included to facilitate query interaction with the host programming language. The programmer can select the desired mode of interaction ("UNTIL true", "UNTIL time", or "UNTIL forever") for each query.

3.5 Event Query Processing

The system components of the event query processing engine are illustrated in Figure 2. The event query processing is based on a discrete-time event computation model that involves three stages: Query dispatching, query instantiation, and query execution.

Query Dispatching. The query dispatcher interacts with two major data structures for event query processing: a query list and a service list. The query list holds active queries that are being continuously executed by the system and the service list holds the thing-service pairs that are streaming events to query processor. A fresh system starts with both lists being empty. Each new query subscription initiates a query processing service with the Dispatcher process. The Dispatcher goes through the parsing process to extract the required thing-service pairs, builds an event expression tree (Figure 4), and creates a query object from the query string. For each thing-service pair, the Dispatcher checks whether the smart service exists in the service list; and it creates a new connection to the host where the service runs. The Dispatcher completes by updating the query list and the service list with the new query object and new smart services, respectively.

Query Instantiation. In this stage, query runtime environments are prepared and a query execution thread is instantiated for each new query. Specifically, two types of critical data structures, namely, the *base dynamic relation* and the *dynamic view* are created. A *base dynamic relation* is created for each "thing" given the extracted thing-service pairs and it can be shared by different queries. In contrast, a dynamic view is only accessed by a particular query because it's created for the event expression tree of that query (Figure 4). The data stored in the dynamic view represents the intermediate computation results and query results specific to a query. A query execution thread is instantiated after the dynamic relations are created.

Query execution. Query execution involves repeated evaluation of the event predicate following a discrete time model. When a query starts being executed, the involved smart services keep running until explicitly terminated or their ending conditions are satisfied. An event predicate is evaluated in a bottom-up fashion starting from leaf nodes of its event expression tree. In detail, query processor receives event computation results of smart services through the discrete time sampler in an asynchronous manner. The discrete event sampler (Figure 2) is an interface program in the query processor that runs with a timer in discrete time steps and samples all of the smart service states consecutively. Then, all of the query execution threads are signaled for the existence of the new data, hence they enter the next execution stage if they have completed the previous one. A query execution thread always retrieves the sampled event state, rather than the most recently arrived one. In this way, the timestamps of different atomic events in a composite event query are kept close to each other for temporal allignment.

Intermediate operator nodes in the query execution tree take columns of base dynamic relations (i.e. computation results of smart services) and/or columns of dynamic views (i.e. computation results of the child operators) as input. The results of the entire event predicate are output from the root node of the event expression tree (stored in the last column of the dynamic view table). Once the event predicate is evaluated to be true, the query processor activates the event-handler program to execute the predefined actions.

When a the execution of a query ends, its subscriptions to the different event streams are terminated accordingly. Unused smart services are periodically disposed from active service list. Each active service object holds a list of query clients currently subscribing to the service. Smart services with an empty query subscription list may be disposed. A new query is added to the related service subscription lists before it enters the active query list to avoid the required smart services being removed while running. This also avoids locking of the entire service list when a query stops being executed. Moreover, since the system does not immediately terminate the smart service connections after the query leaves, it allows new queries to keep the service alive, which reduces the service subscription overhead.

3.6 Query Services

ThingStore provides two powerful services for developers to develop applications atop, namely, the service discovery service and the event query service.

The service discovery service supports a browsing interface to enable the application developer to query the metadata of "things" and smart services in order to discover potentially useful "things." The retrieved device IDs can then be used in the formulation of the event query. More advanced applications can also query the metadatabase at run time in order to dynamically select relevant "things" based on the end user's interests.

Event query service allows applications to submit EQL queries to *ThingStore*, and optionally receive event responses through an asynchronous interface. This service is composed of subscription interface functions, "*subscribe*" and "*unsubscribe*;" and an event callback function. The *Subscribe* function allows applications to submit a new query with an EQL string, or a query ID passed as an argument if it is a stored query already registered with the service. Since EQL queries can be used by multiple users, additional context information for the event handler is passed as another argument. The continuous queries are associated with sessions, and can be terminated through the *unsubscribe* function. The event callback function is used to enable asynchronous interaction between the application and the query processor in *ThingStore*. To use this service, the application includes a callback entry in its code and implements the event-handler (specified in the SELECT clause of the query) to call the application at this callback entry upon event detection. Similarly, the application needs to also include a callback entry for query-termination notification. When a query terminates according to the condition specified in the UNTIL clause, *ThingStore* calls the application at the query-termination callback entry to notify query termination without the *unsubscribe* signal being sent.

The query interface becomes a powerful tool when coupled with the EQL functionalities such as event handler and the UNTIL clause. A query's lifespan can be controlled according to *ThingStore* clock by using UNTIL, or it can be specified by the application logic based on a timer or an event trigger. Asynchronous event feedback enables listening to the event without loops or blocking. This allows application to run its own logic to trigger more events or subscriptions without excessive delay.

The following Function 1 and 2 are examples of application-side subscription calls without a busy wait or loop. In the first example, the $WaitForAnEventAndProceed()$ function subscribes to a stored query that has been pre-defined in the database, and waits for an event for a maximum of $maxWait$ milliseconds. In case of a true state of the event, the function stops waiting and returns. In order to achieve this functionality, a callback contract class, $ServiceCallback$, is implemented by the application, and an instance is given to the service object. Wait operation is performed through a reset event object, which allows re-activation of the waiting thread from the callback class. Event response is received by the *OnEvent* function, which signals the reset event in the main thread. As can be seen from this example, waiting for an event while also checking for the timeout does not require any inefficient loop structures. In the second example, the client subscribes to two queries at a time and proceeds only if the total number of true event states reaches ten. As can be observed from this example, the application logic to trigger an event can be easily implemented into the callback class.

Function 1: Waits for a single event for maximum given time and proceeds.

```
SmokeQuery: SELECT Callback FROM C0 WHEN
     C0.SmokeDetector WITHIN 2 UNTIL FOREVER

void WaitForAnEventAndProceed(int maxWait){
   ev = new AutoResetEvent(false);
   cb = new ServiceCallback(ev);
   service = new ClientServiceContract(cb);

   service.SubscribeQuery("SmokeQuery");
   ev.WaitOne(maxWait);
   service.Unsubscribe();
}

class ServiceCallback : IClientCallbackContract
{
```

```
    AutoResetEvent m_autoEvent;
    ServiceCallback (AutoResetEvent ev)
       {  m_autoEvent = ev; }
    void OnEvent()
       {  m_autoEvent.Set(); }
    void OnQueryTermination()
       {   m_autoEvent.Set(); }
}
```

Function 2: Waits for a total of ten events from two different subscriptions and proceeds.

```
void WaitForTenEventsFromTwoQueriesAndProceed(){
    ev = new AutoResetEvent(false);
    cb1 = new ServiceCallback(ev);
    service = new ClientServiceContract(cb1);
    cb2 = new ServiceCallback(ev);
     service2 = new ClientServiceContract(cb2);

     service.SubscribeQuery("InfiniteQuery1");
     service2.SubscribeQuery("InfiniteQuery2");
     ev.WaitOne();
     service.Unsubscribe();
     service2.Unsubscribe();
}

class ServiceCallback_10times : IClientCallbackContract
{
    AutoResetEvent m_autoEvent;
    int cnt;
    ServiceCallback (AutoResetEvent ev)
       {  m_autoEvent = ev; cnt=0; }
    void OnEvent(){
       cnt++;
       if(cnt >= 10)
         m_autoEvent.Set();
    }
    void OnQueryTermination()
       {   m_autoEvent.Set(); }
}
```

3.7 Comparison with traditional DBMS

The similarity between the *ThingStore* environment and the traditional database application development is illustrated in Figure 6 to give some insight into the proposed IoT framework. We can view "things" in *ThingStore* as a special class of storage devices and the smart services as user-defined thing operators. These operators are analogous to access methods designed to make disk access more intelligent. Traditional database query processing is based on search key values (e.g., phone number is 407-458-9394), whereas an EQL query is based on event search. Much like a database management system that filters out unqualified data records in query processing, the EQL processor in *ThingStore* returns only events specified in the EQL queries. IoT apps, threfore, can retrieve only interesting events for furhter decision making.

ThingStore allows application developers to develop IoT apps much like traditional database applications that are conveniently built atop a commercial database management system. They do not need to have expertise in video and sensor computing, and can instead fully focus on the high-level semantics of their applications. *ThingStore* also provides a clear division of responsibility between the IT (Information Technology) department and the other departments it supports. As an example, IT would develop, deploy, and maintain *ThingStore*; while the engineering and other departments may contribute the smart services and applications. IT may also offer IoT software development and consultation service to departments that are less technology savvy.

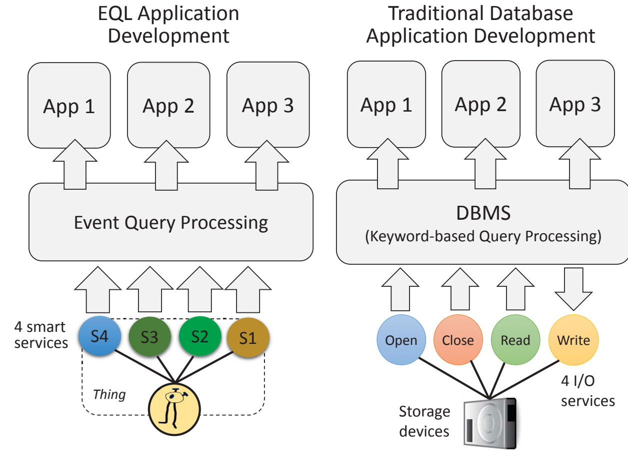

Figure 6: Similarity between *ThingStore* and traditional database application development environment.

4. SYSTEM PROTOTYPE AND EXPERIMENTAL STUDY

The system prototype is implemented using Microsoft .NET Framework. The WCF library is used for the communications among thing servers, the StoreBuilder and the query processing server. Microsoft SQL Server is used as the database server of *ThingStore*'s market place. The StoreBuilder was developed in Unity environment to provide an intuitive 3D GUI to manage "things" and services.

As a proof of concept, we developed a smoke detection service *SmokeDetection* for a video camera named *S4*, and added both of them to the market place using the StoreBuilder. We tested the functionalities of the system using a number of event queries and they all successfully detected the events and triggered the specified event-handlers. An example of the tested query is "**SELECT** *Display* **FROM** S4 **WHEN** S4.SmokeDetector **WITHIN** 5 minutes" which queries for the event "smoke is detected by camera S4 within a 5 minutes time window", where *Display* is an event-handler program that starts to stream video to the application when the event is detected. Figure 7 shows a snapshot of the system prototype and the result of this test query.

A controlled event generator is implemented for precise system performance analysis under various loads and parameters. In order to create a controlled test environment to measure end-to-end parameters between service and the query subscriber, both of them are combined in a unified test program. The system and the test programs are located on a single computer with a 3.47GHz Intel Core i7 X 990 CPU, 24GB of RAM and 64-bit Windows 8 operating system. Both the maximum and minimum CPU frequency are set to 80% in order to avoid unwanted effects of dynamic frequency scaling to be seen in the analysis. Unless otherwise specified, event sampling period is set to 15 ms, which is the minimum system timer period available.

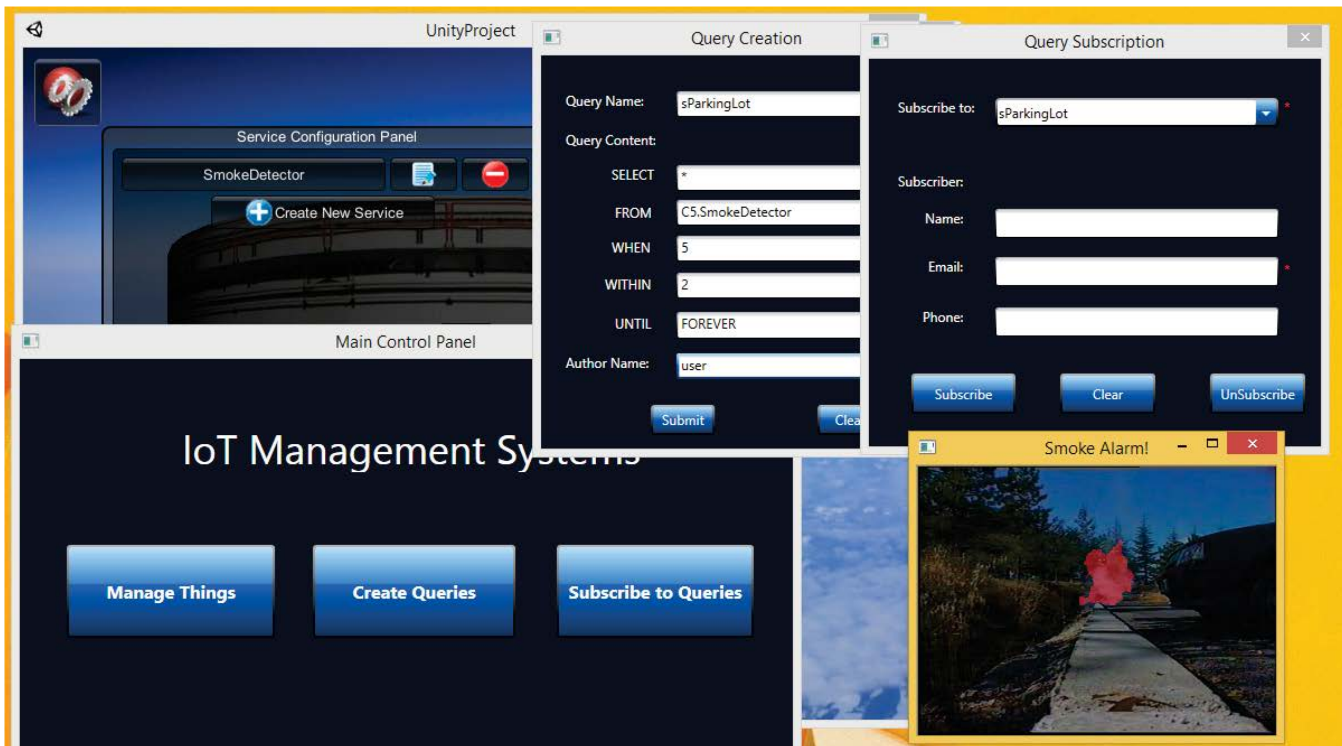

Figure 7: A snapshot of the *ThingStore* system prototype. Upper left: StoreBuilder. Upper right: video streaming from video camera *S4* when specified event is detected. Middle row: main control panel. Lower left: query subscription panel. Lower right: query creation panel.

The first setup intends to measure the response time and the utilization of the system under different loads. The test program continuously increases the number of query subscriptions. The event predicate starts with false state and turns into true state. Each query subscription is for a different event and they are single event queries. Response time (delay) is measured as the time difference between the first true event signal from the smart service and the arrival of event notification to the query subscriber. Throughput is measured as the total number of event notifications when the query's event predicate is set to constant true. Median value of ten test simulations are presented.

Fig. 8a shows response time of the system for a new query while 0-1000 active queries are being processed and Fig. 8b shows throughput of this number of active queries. As can be observed from linear increase in throughput, the system can successfully serve up to 400 concurrent subscriptions. Further increase of the number of subscriptions goes beyond the processing capability of the hardware. Similar effect can also be observed from Fig. 8a. While the median value of delay is usually below 15 ms for up to 400 subscriptions and it goes above 15 ms with more subscriptions. In the latter case, when the query processing delay is longer than the event sampling period, system does not accumulate more event evaluations, but runs the next event sample instead. Hence, increase in delay is approximately linear when the number of queries exceeds the processing capability. However, it is still unpredictable due to operating system's scheduling policy. From Fig. 8b, we can also observe that there is an increase in the number of queries being processed until CPU is fully loaded.

System delay and throughput are also tested under different event sampling frequencies. Fig. 8c shows mean delay of 250 atomic event queries under different frequency configurations. As can be observed from the graph, the average delay is always about half of the event sampling period. From our experiments, it is also observed that the range is usually between zero and sampling period. This experiment shows that the major source of delay is the event sampling frequency which is a controllable parameter. The second analysis, as illustrated in Fig. 8d, shows the throughput for 1200 concurrent queries. While the sampling period is inversely proportional to the number of responses, 15-30 ms area is less stable due to the excessive load in the server. From those experiments, it is observed that event sampling frequency is an important parameter that enables the adjustment of computational resources according to the platform requirements.

Finally, the system is tested for the complexity of queries. Initially, singleton queries involving "And" operation of 1000 events are tested for the delay. However, no difference was observed because query processing time was still negligible over 15 ms event sampling period. In order to simulate system behavior further for longer query executions, long-running queries are artificially created so that they execute busy loops in a given timespan. Fig. 8e and 8f show the response time and throughput of the system for different query execution lengths. Results are median value over 10 simulations. While delay is for a single query, throughput is measured for 20 simultaneous queries.

As can be observed from Fig. 8e, median delay of a long-running query can be estimated as $1.5T_x + 0.5T_s$, where T_x

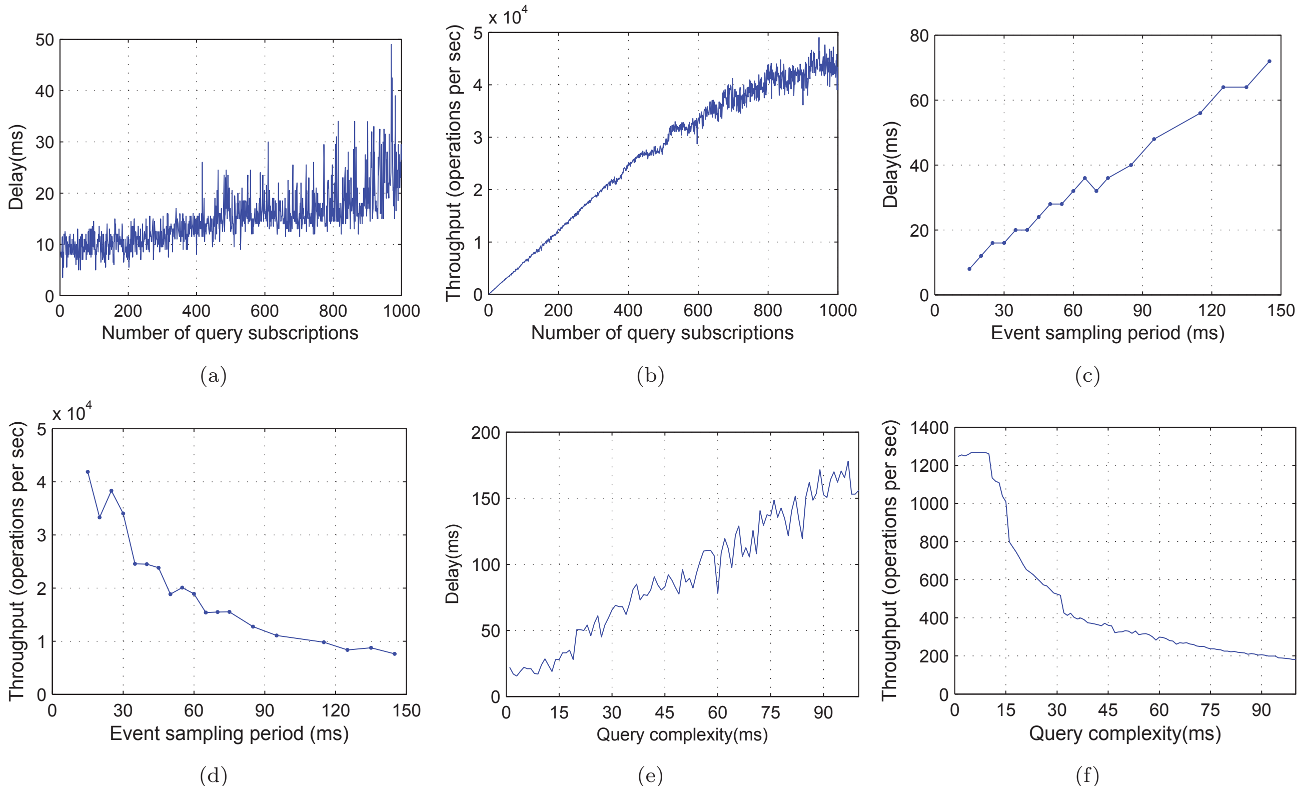

Figure 8: Performance analysis of event query execution. Delay and throughput of the system under different system loads (a,b), event sampling frequencies (c,d), and query complexities (e,f).

is the query execution time and T_s is event sampling period. In the first term, a half of T_{ex} comes from the previous cycle of query execution that does not result in notifications, and one T_{ex} is caused by query processing that results in notifications. The second term is the average delay for arrival of the event signal to the query processing thread.

From Fig. 8f, we can observe that the throughput over query execution time is almost constant between 0-10 ms interval where CPU is not fully utilized and query complexity is smaller than the event sampling interval. With full utilization of the hardware, throughput becomes inversely proportional to the query complexity because each query thread executes in period of the query execution time rather than the event sampling period.

5. RELATED WORK

ThingStore is closely related to two research areas, namely IoT architectures and data stream processing. In this section, we give a brief review of related work in these two areas.

Recent years have witnessed an explosion of IoT platforms. Some of the research proposals target specific application domains. For example, [26] presented an integrated service framework focusing primarily on a smart-house environment. They aimed to provide a complete software chain that supports the design, creation, publishing, deployment and execution of wrapper services. In [9], the authors propose an architecture for smart office application. Their primary goal is to expose the wireless sensor and actuator networks to the Internet as a web service. Other examples include IBM EPCglobal Network which targets RFID-enabled smart supply chains, Texas Instruments' IoT initiative that focuses on smart grid [19] etc.

A number of research works aim to provide a holistic solution for general IoT architecture. [27] discusses the limitations of a three-layer structure, and proposes a five-layer architecture. In this design, the existing Application-Network-Perception layered structure is extended with the business layer and the processing layer. Another extension of the three-layer architecture is proposed in [12], in which an adaptation layer is added between the network layer and the perception layer. More recently, [24] presented a more detailed layered design that provides various levels of abstraction in an effort to address issues such as scalability, heterogeneity, and interoperability of future IoT. Authors of [13] approach the IoT architectural problem from a service modelling perspective. In this work, a semantic modeling approach is adopted to integrate heterogeneous physical objects with the digital world and make them accessible on a large scale. A similar proposal is the SENSEI project [22] which focuses on sensor descriptions and sensor data modelling.

Several other works present novel design and implementation. For example, [2] uses an event-driven service oriented architecture (e-SOA) to expose object's functionality in the form of web services. An adapter-based approach is adopted to address the heterogeneity of connected devices and semantic rules are provided to facilitate policy-based service access. In [15], the author presents a cloud-based framework for world-scale implementation of IoT. A cloud implementation using Aneka, which is an application development

Platform-as-a-Service (Paas) combining resources from public and private clouds, is presented in this paper. The reader is referred to [25] and [23] for a comprehensive survey of the architectural designs.

Our architectural design differs from previous proposals in profound ways. We address the IoT challenge from the software development perspective and the proposed *ThingStore* is an advanced app store that fosters the cooperation and innovation of thing provider, software developer, and end user. To the best of our knowledge, we are among the first to target an integrated platform for IoT software development, if not the only one. The only research work that is conceptually similar is [20], which promotes the concept of an IoT application market place to foster open participation and innovation of software developers. However, the authors' analogy with the smartphone app store limits its scope of innovation. In contrast, *ThingStore* offers features more than an application market: it also features an advanced computational paradigm that uses a query processing engine to process and handle complex event queries.

Data Stream Processing is another area related to the *ThingStore* project. Stream processing systems can be classified into two general categories: data stream management systems (DSMSs) and complex event processing systems (CEP systems) [11]. DSMS is an extension to traditional DBMS for analyzing data streams coming from multiple data sources [4, 5, 18, 17, 29]. Similar to DBMS, most DSMSs also process data with a series of operations defined by relational algebra. For example, [18] designed an acquisitional query processor to process sensor data streams according to user-specified queries. The SQL-like queries can be formulated to filter data with predicates, and/or to transform the sensor data using aggregate operators. On the other hand, the differences between DSMS and DBMS are also apparent in terms of timeliness, continuousness, query data scope, processing mechanism, etc.

CEP systems model the inbound data streams as sequences of real-world event notifications and aim to identify interested event patterns through high-level semantic composition of the low-level events. Instead of providing the numerical computation results on the data streams, which is the typical practice of DSMS, the CEP systems return in the form of notifications upon detection of pre-defined events. Examples of CEP systems are [1, 6, 7]. The reader is referred to [11] for a comprehensive review of such systems. A major focus of CEP systems is to address the limitations of DSMSs in detecting complex patterns data streams by using complex event specification languages [10].

DSMSs and CEP systems mostly are designed to target specific application domains, such as wireless sensor networks [18, 17, 29], environment monitoring [8], network traffic monitoring [14], etc. There have been few works approaching the design of IoT with such data stream processing techniques. Our approach extends the stream processing framework to overcome problems associated with "thing" hetorogenity and provide a unified software platform for IoT application development and Internet-scale event query processing.

6. CONCLUSIONS AND FUTURE WORK

We introduce an architecture for IoT applications development and deployment. This proposed environment has the following advantages:

- The approach based on smart services provides a unified binary abstraction to address the challenge associated with heterogeneity of devices. The software developers can develop their IoT applications without having to deal with differences in the underlying devices (e.g., cameras vs environmental sensors).

- The continuous query processing model based on EQL (Event Query Language) enables "things" to collaborate on common tasks through applications, regardless of their physical locations in the IoT ecosystem. This is an improvement over current intranet of things deployed in silos to support different IoT applications.

- Decoupling of smart services from the application logic allows pushing thing-specific computation (such as computer vision computation associated with a camera) closer to the live data source (i.e., "things"). This strategy avoids network traffic and therefore achieves a scalable architecture for large-scale deployment of IoT applications.

- *ThingStore*, based on the above concepts, enables collaboration on software development and cost sharing of the expensive IoT infrastructure.

We demonstrate the above concepts through a system prototype and present experimental results to illustrate the effectiveness of *ThingStore* as an architecture for IoT application development and deployment. As a system, *ThingStore* provides a computation hub that links together human,"things," and computer software in a cyber-physical lifecycle to enable fusion of human and machine intelligence to accomplish some common goal. Not only human, but also "things," may adjust the physical world. New changes in the physical world may, in turn, incur new event detections and therefore initiate another cycle of this ecology-inspired computational lifecycle.

Our future work will focus on query optimization techniques for better execution of many queries concurrently (i.e., inter-query parallelism). This includes scheduling techniques to facilitate sharing of computation and storage access among the active queries. It is also desirable to develop a search technique for finding relevant "things" in the IoT ecosystem.

7. ACKNOWLEDGMENTS

This work is partially supported by NASA grant, grant number: NNX14AQ29A. Any opinions, findings, conclusions or recommendations expressed in this materials are those of the authors and do not necessarily reflect the views of NASA.

8. REFERENCES

[1] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008.

[2] S. Alam, M. Chowdhury, and J. Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6, Nov 2010.

[3] J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. Technical report, Rochester, NY, USA, 1994.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[5] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[6] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, January 2007.

[7] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM.

[8] K. Broda, K. Clark, R. Miller, and A. Russo. *SAGE: a logical agent-based environment monitoring and control system*. Springer, 2009.

[9] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi. Architecture and protocols for the internet of things: A case study. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 678–683, March 2010.

[10] G. Cugola and A. Margara. Tesla: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[11] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[12] G. Dai and Y. Wang. Design on architecture of internet of things. In D. Jin and S. Lin, editors, *Advances in Computer Science and Information Engineering*, volume 168 of *Advances in Intelligent and Soft Computing*, pages 1–7. Springer Berlin Heidelberg, 2012.

[13] S. De, P. Barnaghi, M. Bauer, and S. Meissner. Service modelling for the internet of things. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 949–955, Sept 2011.

[14] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In W. Lee, L. MÍę, and A. Wespi, editors, *Recent Advances in Intrusion Detection*, volume 2212 of *Lecture Notes in Computer Science*, pages 85–103. Springer Berlin Heidelberg, 2001.

[15] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, 2013.

[16] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, Jan 2010.

[17] S. Madden and M. Franklin. Fjording the stream: an architecture for queries over streaming sensor data. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 555–566, 2002.

[18] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, Mar. 2005.

[19] O. Monnier. A smarter grid with the internet of things. *Texas Instruments White Paper, Oct*, 2013.

[20] D. Munjin and J.-H. Morin. Toward internet of things application markets. In *Proceedings of the 2012 IEEE International Conference on Green Computing and Communications*, GREENCOM '12, pages 156–162, Washington, DC, USA, 2012. IEEE Computer Society.

[21] C. Perera, A. B. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *CoRR*, abs/1305.0982, 2013.

[22] M. Presser, P. Barnaghi, M. Eurich, and C. Villalonga. The sensei project: integrating the physical world with the digital world of the network of the future. *Communications Magazine, IEEE*, 47(4):1–4, April 2009.

[23] O. Said and M. Masud. Towards internet of things: Survey and future vision. *International Journal of Computer Networks*, 5(1):1–17, 2013.

[24] C. Sarkar, S. Nambi, R. Prasad, and A. Rahim. A scalable distributed architecture towards unifying iot applications. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 508–513, March 2014.

[25] D. Singh, G. Tripathi, and A. Jara. A survey of internet-of-things: Future vision, architecture, challenges and services. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 287–292, March 2014.

[26] X. Su, R. Svendsen, H. Castejon, E. Berg, and J. Zoric. Towards an integrated solution to internet of things - a technical and economical proposal. In *Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference on*, pages 46–51, Oct 2011.

[27] M. Wu, T.-J. Lu, F.-Y. Ling, J. Sun, and H.-Y. Du. Research on the architecture of internet of things. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 5, pages V5–484–V5–487, Aug 2010.

[28] X. xiaoli, Z. Yunbo, and W. Guoxin. Design of intelligent internet of things for equipment maintenance. In *Intelligent Computation Technology and Automation (ICICTA), 2011 International Conference on*, volume 2, pages 509–511, March 2011.

[29] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3):9–18, 2002.