Application Caching for Cloud-Sensor Systems

Yi Xu
CISE Department
University of Florida
Gainesville, FL32611, USA
yixu@cise.ufl.edu

Sumi Helal
CISE Department
University of Florida
Gainesville, FL32611, USA
helal@cise.ufl.edu

ABSTRACT

Driven by critical and pressing smart city applications, accessing massive numbers of sensors by cloud-hosted services is becoming an emerging and inevitable situation. Naïvely connecting massive numbers of sensors to the cloud raises major scalability and energy challenges. An architecture embodying distributed optimization is needed to manage the scale and to allow limited energy sensors to last longer in such a dynamic and high-velocity big data system. We developed a multi-tier architecture which we call Cloud, Edge and Beneath (CEB). Based on CEB, we propose an Application Fragment Caching Algorithm (AFCA) which selectively caches application fragments from the cloud to lower layers of CEB to improve cloud scalability. Through experiments, we show and measure the effect of AFCA on cloud scalability.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Communications

Keywords

Cloud-sensor systems; Cloud computing; Application caching.

1. INTRODUCTION

Considering the emerging smart city developments and the massive number of sensors, devices and applications to be integrated into the cloud – the expected platform of choice, we argue that any proposed architecture for an efficient, scalable cloud-sensor operation must address the following challenges.

Cloud Scalability: Extensive interactions between cloud services and sensors could pose huge challenges to the cloud scalability. Connecting millions of sensors directly to the cloud is extremely demanding on cloud resources. It results in expensive cloud "attention", not only per sensor, but per each sensor duty cycle. For example, if sensors push data once every minute, millions of sensors will produce billions of sensor-cloud interactions, daily. As a result, the cloud economies of scale per sensor will not stand.

Sensor Energy Constraint: Unlike cloud resources with elastic supply, sensor devices cannot be provided on demand, and many of them are extremely vulnerable to power drainage. In smart city scenarios, a sensor may be queried by hundreds of applications each of which requires constant sensor readings. Without optimization, sensors' energy could be depleted rapidly, failing services and making them unreliable and unavailable.

Thus, a structural basis for optimizing the cloud's use of sensor hardware is needed or cloud-sensor systems will not be dependable. To achieve this, we adopt a three-tier architecture that enables cross-layer optimizations [1]. Based on this architecture, we present a bi-directional waterfall optimization framework under which several optimizations can be carried out to achieve greater cloud scalability and energy-efficiency of sensors. We focus on one optimization in this paper – application caching and presents experiments and analysis of the algorithm.

2. CEB ARCHITECTURE OVERVIEW

In [1], we proposed a three-tier Cloud, Edge and Beneath (CEB) architecture whose layers are depicted in Figure 1. CEB is built on top of Atlas [2] that implements the service-oriented device architecture (SODA) model [3] and automates the process of sensor integration through a sensor platform and middleware which are integrated into the cloud availed for use by cloud applications. Because of the scope of this paper (cloud-to-edge application caching), we only explain the cloud and edge layers.

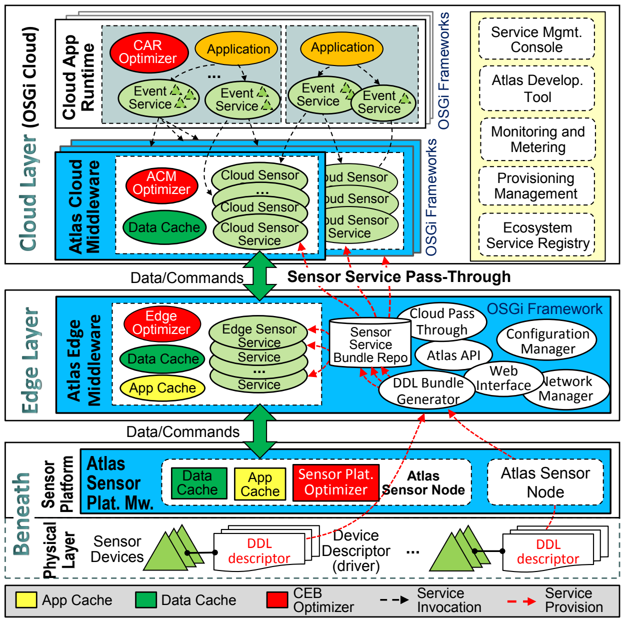


Figure 1. Overview of the CEB architecture

The edge runs one prong of the Atlas middleware which uses OSGi [4] as its basis to provide service discovery and configuration. The middleware includes a bundle generator, which when contacted by an initializing Atlas sensor platform, creates a pair software bundles for each sensor: 1) *edge sensor service* to be hosted at the Atlas edge middleware, 2) *cloud sensor service* to be uploaded to the Atlas cloud middleware. The pair of services communicate with each other enabling interlayer data passing.

The Cloud layer is built on OSGi Cloud [5] in which application is composed by loosely-coupled modules as OSGi services hosted at a distribution of cloud nodes (host OSGi framework) via cloud-wide discovery, configuration mechanisms. To help elaborate our work in this paper, we explain two specific components in the cloud layer most relevant to application caching.

Atlas Cloud Middleware (ACM): For every edge, there exists a corresponding ACM at the cloud. It hosts the cloud sensor service (passed from the edge) to be subscribed to by other cloud services. ACM acts as the cloud gateway to the lower layers, and meanwhile, it hosts the most basic "clouding" of sensors based on which sensor-based cloud applications can be built.

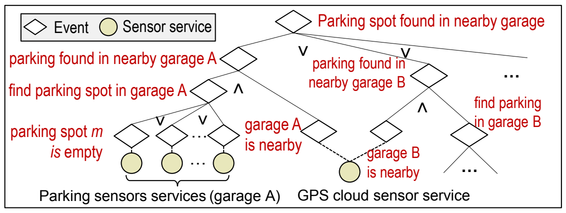
Cloud Application Runtime (CAR): The container where application-specific services are deployed and managed. Applications make invocation to the sensor services at the ACM to acquire raw sensor readings from the physical deployment.

CEB supports different application models that program smart space sensors into different levels of abstractions (e.g., events, activities, and phenomena). In this paper, we choose a specific application model E-SODA that abstracts sensor data into events.

2.1 E-SODA Application Model

E-SODA presents a particular service type *Event Service* which subscribes to and invokes the sensor services at the ACM to implement event-level abstractions of sensor data. An event service listens to the occurrence of a particular event which is a logical expression over sensor values (event composite grammars are shown in Equation 1) by processing its *event representation tree* (ERT) (Figure 2 illustrates an example).

$$\begin{aligned} E &= sensor\,(value) \mid sensor\,[a,b] && \text{(atomic event)} \\ &= \mid \sim E && \text{(negation)} \\ &= \mid E \vee E \mid E \wedge E && \text{(or/and)} \\ &= \mid E\,?E : E && \text{(condition operation)} \\ &= \mid E * time * E && \text{(sequence)} \\ &= \mid \{E\} && \text{(scope block)} \end{aligned} \tag{1}$$



Event
Sensor service
Parking spot found in nearby garage
parking found in nearby garage A
find parking spot in garage A
parking spot *m* is empty
∨
∧
parking found in nearby garage B
...
find parking in garage B
garage A is nearby
garage B is nearby
Parking sensors services (garage A)
GPS cloud sensor service

Figure 2. An example event representation tree (ERT)

We also introduce an application-specific relaxation operator, the time/frequency modifier (TFM), which is specified as follows:

$$\begin{aligned} TFM &=< W, I_e > \\ W &= \text{nil} \mid \text{date/time} - \text{date/time} \mid \text{time} - \text{time} \\ I_e &= \text{Interval (\# of seconds) between two successive evaluations} \end{aligned} \tag{2}$$

W is a time window in which the affected event is evaluated with interval I_e. TFM allows programmers to relax event evaluations based on user specific demands or sensor semantics. It also enables aggregate query (piggyback) to be performed on multiple sensors connected to the same edge and applied to the same TFM.

3. BI-DIRECTIONAL WATERFALL OPTIMIZATION FRAMEWORK

In plain cloud-sensor systems, applications reside in the cloud which request and process data originating from the physical layer. We extend the plain model and propose a *bi-directional waterfall optimization framework* [7] which allows not only data to move upward but also allow application (fragment) to move downward and get cached at lower layers. In E-SODA, the cacheable application fragments are *events*. A cached event is evaluated at the layer it is cached to and its event value is pushed back to its upper layer only when it changes (i.e., *selective push*). With application caching, the workload on the cloud can be dispersed or diffused across a group of edges or even sensor platforms. Therefore, *cloud scalability* as one challenge of cloud-sensor systems can be addressed effectively. Metaphorically, the bi-directional waterfall approach allows for most interesting data to flow up and for most curious applications to propagate down. Consequently, a local view of both data and applications and hence their interactions and interplays can be monitored and analyzed at any layers of the CEB, and therefore a number of optimization strategies can be implemented to optimize the energy consumption of the sensors. In this paper, we focus on one of the algorithms – Cloud-to-Edge Application Fragment Caching Algorithm (AFCA-1) which aims to improve the cloud scalability.

4. CLOUD-TO-EDGE APPLICATION FRAGMENT CACHING ALGORITHM

Caching application from the cloud to the edge layer reduces the workload of processing events on the cloud servers. Given the elasticity of the cloud, such workload reduction will shrink or slow down the growth of the cloud dimension in terms of the number of reserved cloud instances. Consequently, cloud scalability is improved. Meanwhile, caching events to the edge layer consumes resources in edge servers. Unlike the cloud, edge servers have limited resources and cannot unlimitedly cache applications from the cloud. In this sense, we propose the AFCA-1 algorithm to select the application fragments (i.e., events) from the cloud to cache down at the edge layer with the the Objective: *to minimize the cloud dimension (i.e., number of cloud instances)*, under the Constraint: to *stay within the resource limitations in edge servers*.

4.1 Implementation of Application Caching

Figure 3 depicts our implementation of caching applications from the cloud to the edge layer. In E-SODA, *event services* make for an ideal application fragments to cache. The machine instances in the cloud and edge layer all install the same OSGi environment, and after an event service is cached at the edge layer, it can automatically re-bind to the edge-level sensor services (peers of the cloud-level sensor services it subscribes to) via the *soft service binding* implemented by CEB [1] with minimum overhead. In addition, for any event (e.g., E_2 in Figure 3) cached to the edge layer, a "shadow service" of the cached event is correspondingly created in the ACM acting as a proxy of the cached event (e.g., E_2) to its consumers (e.g., E_1).

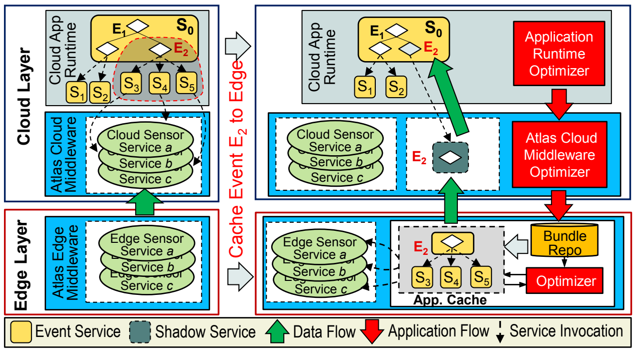


Figure 3. Implementation of application (event) caching

4.2 Understanding the Effect of App Caching

In our design, all cloud instances install the same AMI instantiated on a standard machine type *m1.medium* whose resource capacity is shown in Table I. For all instances, we adopt a common maximum memory and CPU utilization denoted as γ_1 and γ_2. Of all the components in the cloud layer, caching applications to the edge primarily affects the cloud resource usage in two components: CAR and ACM. In [6], we found that for both ACM and CAR, the dominant resources that determine the cloud dimension can be either *processing* or *memory*. For ACM, when processing resource becomes dominant, caching events to the edge reduces communication between CAR and ACM and between ACM and edge servers for evaluating the cached event. Thus, processing resource in ACM is saved due to reduced data transmission. On the other hand, event caching causes the addition of a shadow service to the ACM which introduces additional processing cost due to sending and receiving the value of the shadow event. When memory becomes dominant, caching event to the edge increases the memory cost because of the additional shadow service. For CAR, when processing becomes dominant, caching event to the edge reduces the processing resource consumed by the cached services. When memory becomes dominant, event caching releases the memory resource allocated for the cached service. The above cause-effect analysis between dominant resource usage at the ACM and CAR directly influences and guides the AFCA-1 algorithm.

Table I. Cloud instance resource capacity [6]

ECU	Memory	Storage	Bandwidth
2	3.75 GB	160 GB	–

4.3 Algorithm Description

AFCA-1 selects a set of events in the cloud applications to cache at the edge to maximize the reduction of the overall dominant resource usages in all affected cloud components. We describe the AFCA-1 (run at the cloud side) as follows. For each event in a cloud application, it calculates the overall *Benefit* of caching that event to the edge layer, and based on the calculation a group of events is selected to send to the edges for final caching decisions.

```
AFCA-1 (run at the Cloud App Runtime)
1. for each Event in an application construct its ERT as T
2. Partition T into areas (sensors of an area connect to the same edge);
3.   for each area
4.     for each event $E$ calculate its Benefit($E$); endfor
5.     selectedEvents = EventSelect(area);
6.     Send selectedEvents to edge and wait for the response;
7.     Receive response (the events approved to cache) from edge;
8.   endfor
9. endfor
```

EventSelect method is to achieve the maximal total *Benefit* for the selected events while ensuring that at most one event is chosen at any branch in the *ERT*. *EventSelect* returns a *selectedEvents* which is a list of vectors for all selected events <*eventId*, *Benefit*, C_{edge}> ordered by *Benefit* (decreasing order). *Benefit(E)* is calculated as $(S_{AR}+S_{ACM})/C_{edge}$, where S_{AR} (Equation 3) and S_{ACM} (Equation 4) denote respectively the saving of resource usage for CAR and ACM after caching event *E* to the edge, and C_{edge} (Equation 5) denotes the extra resource usage added to edge.

$$S_{AR}=\begin{cases} 1/\gamma_1 \times \dfrac{\sum_{es\in ES} mem(es)}{M_c} & \text{(memory critical)} \\ 1/\gamma_2 \times \dfrac{\sum_{es\in ES} cpu\_time(es)}{elapsed\ time} & \text{(processing critical)} \end{cases} \quad (3)$$

ES denotes the event services that compose event *E*. In Figure 3, the *ES* for E_2 is $\{S_3, S_4, S_5\}$. *cpu_time*(*es*) denotes the observed amount of time spent for processing service *es* while the *elapsed time* is the total time of observation. *mem*(*es*) denotes the memory footprint of *es* while M_c is the total memory of the cloud instance.

$$S_{ACM}=\begin{cases} -\dfrac{1}{\gamma_1}\cdot\dfrac{mem_{shadow}}{M_c} & \text{(memory critical)} \\ \dfrac{1}{\gamma_2}\,\dfrac{(cc_s+cc_r)(\sum_{es\in ES} K\cdot f_e(es)-R_c)}{CR_c} & \left(\begin{array}{c}\text{processing}\\ \text{critical, K = 1,2}\end{array}\right) \end{cases} \quad (4)$$

mem_{shadow} denotes the memory footprint of a shadow event service (constant). CR_c is the CPU clock rate of the cloud instance. cc_s and cc_r indicate the number of CPU cycles per packet send and receive respectively. $f_e(es)$ is the event evaluation frequency of *es* (*TFM*) which is also the rate at which *es* sends aggregate queries to the ACM before event caching (will be eliminated after caching). R_c denotes the change rate of the value of event E which is also the rate at which the cloud receives the event value from the edge after event caching. *K* is 1 if *es* only subscribes to public sensors, or 2 otherwise (invoke at least one dedicated sensor).

$$C_{edge}=\begin{cases} \dfrac{1}{\gamma'_1}\times\dfrac{\sum_{es\in ES} mem(es)}{M_e} & \text{(memory critical)} \\ \dfrac{1}{\gamma'_2}\times(C_{eval}+C_{trans}) & \text{(processing critical)} \end{cases} \quad (5)$$

γ_1' and γ_2' denote respectively the maximum memory and CPU utilization allocated for the edge. M_e is the total memory of the edge. C_{eval} and C_{trans} calculate the addition of CPU usage caused by evaluating event *E* (Equation 6) and by the change of data transmissionrate between edge and cloud (Equation 7) respectively.

$$C_{eval}=\frac{\sum_{es\in ES} cpu\_time(es)}{elapsed\ time}\cdot\frac{CR_c}{CR_e}-\frac{\sum_{es\in ES} f_e(es)\cdot(cc_s+cc_r)}{CR_e} \quad (6)$$

$$C_{trans}=\frac{R_c\times cc'_s - req\times(cc'_s+cc'_r)}{CR_e} \quad (7)$$

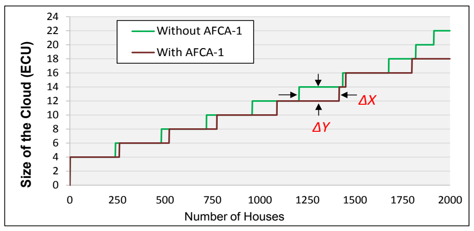
CR_e is the CPU clock rate of the edge server. cc_s' and cc_r' are the number of CPU cycles per packet send and receive by the edge server. *req* denotes the frequency at which the cloud sends request to the edge to evaluate *E* before caching *E* to the edge.

When an edge makes decisions for event caching, events with higher *Benefit* always have higher priority to cache. To do so, for all the cached events the edge maintains a list of vectors <*eventId, Benefit,* C_{edge}> a.k.a. *EBC* whose elements are ordered by *Benefit* in the increasing order. When a new request (i.e., *selectedEvents*) arrives, the edge determines whether to cache or reject the events or replace with existing events based on the *EBC* and its resource availability. The edge-side AFCA-1 is described as follows.

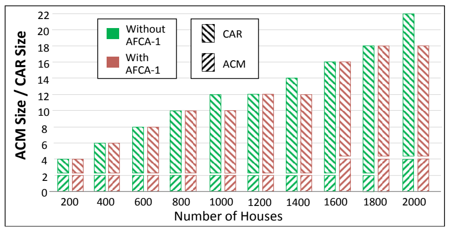
```
AFCA-1 (run at the Edges)
1. Receive selectedEvents from the cloud;
2. for each <eventId, Benefit, $C_{edge}$> in selectedEvents
3.   if edge is not saturated  CL.add(eventId);  // approve caching
4.   else
5.     tSave, tCost, i = 0;
6.     while $i$ < EBC.size and tCost < $C_{edge}$
7.       <e, b, c>=EBC.get(i);  // (i+1)th smallest Benefit in EBC
8.       tSave += b*c;  tCost += c;  i++;
9.     if $i$ < EBC.size and Benefit > tSave/tCost
10.      CL.add(eventId);  // replace existing events in EBC
11.    endif
12.  endelse
13. endfor
14. Cache events in CL, send CL to the cloud and update EBC
```

5. Validation of AFCA-1 Algorithm

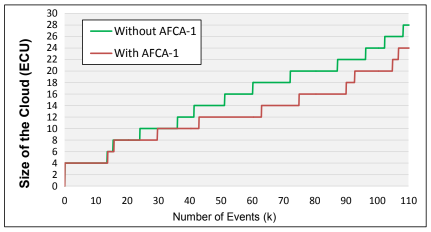
To validate AFCA-1, we set up a prototype of CEB [1] as experimental testbed on which smart home apps are deployed. We synthesized a benchmark for both sensor data and applications



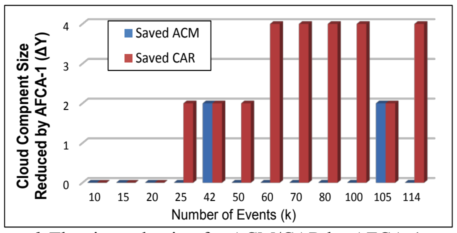
a. Total size of the cloud with horizontal expansion



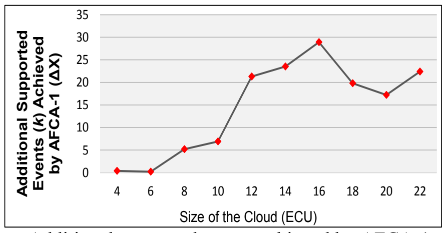
b. Size of cloud components with horizontal expansion



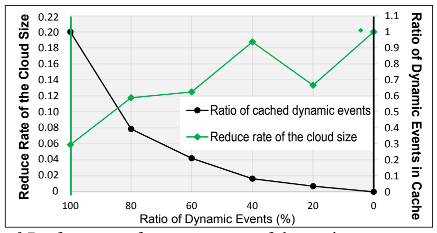
c. Total size of the cloud with vertical expansion



d. The size reduction for ACM/CAR by AFCA-1



e. Additional supported events achieved by AFCA-1



f. Performance for a spectrum of dynamic events

Figure 4. Experiment Results of AFCA-1 Validation

and also classified two types of cloud-sensor system expansion: "*Horizontal*" in which applications are built on dedicated or exclusive sensors (deploying a new application requires installing its associated sensors), and "*Vertical*" where applications are built on pre-existing, sharable sensors which are not bounded to any individual application (adding new services does not require installing additional sensors).

For *horizontal expansion*, Figure 4.*a* shows the growth of the entire cloud and Figure *b* shows the growth of individual CAR and ACM. We observed two offsets: Δ*Y*, the reduction of the cloud size achieved by AFCA-1 to support a particular number of houses, and Δ*X*, the number of additional houses that can be supported in the cloud by AFCA-1. From the results, both Δ*X* and the average of Δ*Y* increase as more houses joining the system. However drop for both offsets occurs when the house number reaches ~1500 and then go back to increase again. This can be explained by Figure *b*, in which AFCA-1 reduces the size of CAR, but increases the size of ACM. This is because, with horizontal expansion, the sensor sampling rate is relatively low (applications rarely share sensors) which makes memory become the dominant resource for ACM. The caching of applications adds to the memory cost to the ACM, so more cloud instances are required to accommodate the increased memory demand.

Under *vertical expansion*, the number of events increases with constant number of sensors. Figure 4.*c* shows the growth of the size of the entire cloud. Figure *d* presents the reduction of the size for each of ACM and CAR achieved by AFCA-1 (Δ*Y*), while Figure *e* shows the additional events that can be held by the cloud achieved by the AFCA-1 (Δ*X*). From Figure *c* and *d*, AFCA-1 does not show any effects until the number of events reaches ~25k. This is because the sensor sampling rate is relatively low when there are few events sampling the sensors which makes the memory become the dominant resource for the ACM, and AFCA-1 thinks the cost of application caching to ACM outweighs the saving to CAR. From Figure *d* and *e*, both the Δ*X* and Δ*Y* offsets increase as the system expands, and stop after the scale reaches a certain level (event number: ~60k, cloud size: 16). This is because most of the edge servers are saturated and event replacement is used in applications caching.

We also validated the reaction of AFCA-1 to event dynamics. We first classified the events into *dynamic event* and *static event* in terms of the rate of event value change. We changed the ratio of the dynamic events in the event set and kept all edge servers in saturate mode, and recorded the results in Figure 4.*f*. From the results, we can see that as the number of the dynamic events decreases, AFCA-1 performance improves from ~6% to ~20%, and the proportion of the dynamic events among all cached events decreases but in a non-liner manner. This result validates that the AFCA-1 performs better when there are more static events in the applications.

6. CONCLUSION

We presented an application caching algorithm (AFCA-1) whose objective is to improve cloud scalability in large-scale cloud-sensor systems anticipated to implement emerging smart urban spaces and smart cities. We also briefly presented the overarching architecture for AFCA-1 - the CEB architecture which embodies a bi-directional data/application caching optimization framework. We presented an extensive evaluation study based on analysis and experimental validation of target performance metrics.

7. REFERENCES

[1] Xu, Y. and Helal, A. 2014. Scalable Cloud-Sensor Architecture for the Internet of Things, Submitted for pub. UF Mobile and Pervasive Computing Lab Technical Report: http://www.icta.ufl.edu/projects/publications/ceb.pdf

[2] King, J., Bose, R., Yang, H., Pickles, S., and Helal, A. 2006. Atlas - A Service-Oriented Sensor Platform: Hardware and Middleware to Enable Programmable Pervasive Spaces. In *Proceedings of the first IEEE International Workshop on Practical Issues in Building Sensor Network Applications*. Tampa, Florida, November 2006, 630-638.

[3] Deugd, S., Carroll, R., Kelly, K.E., Millett, B., and Ricker. J. 2006. SODA: Service Oriented Device Architecture. *IEEE Pervasive Computing*, Volume 5 Issue 3, July 2006.

[4] OSGi 4.2. http://www.osgi.org/Download/Release4V42

[5] OSGi Cloud Computing. https://www.osgi.org/bugzilla/show_bug.cgi?id=114.

[6] Amazon EC2. http://aws.amazon.com

[7] Xu, Y. and Helal, A. 2014. An Optimization Framework for Cloud-Sensor Systems. To be submitted. http://www.icta.ufl.edu/projects/publications/waterfall.pdf