# Towards Automated Building Management through Cooperative Sensor-actuator Networks

Kaisen Lin
UC San Diego
kaisenl@cs.ucsd.edu

Rajesh K. Gupta
UC San Diego
rgupta@ucsd.edu

## ABSTRACT

Sensor-actuator networks require sharing of actuators across multiple applications. Here, simple device arbitration is not enough because actuators have lasting (or irreversible) effects on their environment. For example, actuating a heater not only affects the internal program state of the controlling application, but the nearby temperature as well. This change in external environment often has impact on application-level decision making. We present CAhoot, a preliminary software infrastructure to enable cooperative sensor actuator applications through communication of actuator ranges, and show how it can be applied to automate building-level energy control.

## 1. INTRODUCTION

Sensor network applications have predominantly been passive monitoring applications that collect and sometimes process data. Domain experts then make policy decisions, if any, based on this collected data. In this paper, we focus specifically on joint control of building heating, ventilation, air conditioning (HVAC) and IT power control as an example of a sensor-actuator network application. Studies show that space heating, cooling, and lighting make up 45% of the total energy use [10], and buildings constitute over 70% of power used on the electricity grid [8]. In this context, one policy may be to dim lights and turn off air-conditioning after business hours. However, waiting for a human operator to update a policy when conditions change may be costly. For example, a machine room inside a building may need to be rapidly cooled if it receives a burst in computing workloads to avoid localized heating conditions. Similarly, applicances that do not need to be ON in a home should be turned OFF automatically during an electricity shortage (i.e. demand response).

Sensor-actuator networks are fundamentally different from traditional sensor networks in that they change the nearby environment rather than simply observe it. Naive applications attempting to share actuator usage may end up conflicting on their goals and wasting their collective efforts. If the applications require a precise value to be passed to the actuator, then arbitration of the corresponding actuator will be necessary. In reality, however, many applications instead specify a range for the actuator to achieve desired application goals. For example, Intel recommends using variable fan speeds for CPU temperatures between 30C and 38C [5]. The Occupational Health and Safety Administration (OSHA) recommends 200-500 lux for office environments [3]. The H1N1 virus is most contagious in a building at humidity levels between 35% and 40% [2]. Each of these specifies a range for environmental variables that are controlled directly or indirectly by acutators.

Given this range of specification, it is possible to satisfy requirements from multiple applications by finding the necessary overlap in range requirements. Let's take a look at a simple example. In a home with centralized heating, the living room should have a higher temperature for human activity and a bedroom should have a lower temperature for sleeping. Thus we can write two applications, one for both rooms. A simple way is to check the time, and adjust the temperature based on this. In fact, this is how a static policy would work. However, suppose there is more information, such as who is in which room (based on cell phone presence). Each person can adjust a particular temperature comfort range, and if they overlap the heater can be actuated to a single value. If there is no overlap, then the applications themselves can adjust their requirements if possible. Furthermore, if no other person is in a room, its temperature setting should not matter in terms of human comfort.

In this paper we address the problem of specifying and managing the coordination of shared sensor actuators that affect their surrounding environment.

Applications need to be *actuator-aware.* This allows applications to cooperate rather than compete for actuator use. In particular, applications can change their behavior when they detect another application attempting to use a shared actuator. A centralized policy with knowledge of all actuators and applications can detect and possibly resolve actuator conflicts, but this couples the policy with specific deployment scenarios, which may not be completely known a priori. In the remainder of this paper, we briefly examine related work before describing our design and briefly evaluating it against an automated building management example.

## 2. RELATED WORK

Operating systems typically have sophsticated programming support for controlling hardware peripherals [4]. Hardware virtualization and arbitration is done to avoid unwanted dependencies or race conditions, but this does not take into account dependencies in the physical world. The sharing of actuators fundamentally comes down to resource sharing, where the resources are sensor actuators. There is a rich background in resource sharing, control and management of resources.

The operating system and database communities have explored resource sharing through locking. Mechanisms include semaphores, monitors, and condition variables [15]. An alternative approach is optimistic concurrency control (OCC) [13]. OCC assumes that there is no conflict, and begins writing the shared values. If a conflict is later detected, then the operations are aborted. OCC is particularly good for resources with low contention.

Application control of resources has been seen in the past with disk scheduling [7] and even the machine as a whole [9]. In the disk scheduling work, applications are assumed to know best what their disk access pattern is and can thus give this information to the operating system. However, applications give only suggestions and the operating system still has full control over disk scheduling. In the exokernel, hardware resource control is given to the application, and the kernel enforces a minimal amount of protection among applications.

ECOSystem [16] and ICEM [12] are both systems that provide energy resource management. ECOSystem shares hardware devices by treating energy as a first-class resource through energy "currentcy". ICEM is a device driver architecture that manages energy based on driver concurrency. Energy is saved by inferring usage based on acquisition of locks. Another resource management system specifically for sensor networks is Pixie [14], which allows for resource-aware programming.

Concurrency control, although necessary, is not enough for operation of sensor actuator networks given the coupled and time-bound interactions with the environment. Giving applications full control of sensor actuators may also lead to fairness or security issues, and also burdens application writers with actuator details. The existing energy management schemes provide general purpose techniques for dealing with hardware devices, but further improvements can be made by taking advantage of how sensor actuators are used.

## 3. MOTIVATING EXAMPLE

Consider the operation of a "mixed-use" building. Mixed-use buildings are buildings where their balance of power usage is evenly divided between human comfort and IT equipment. This is in contrast to data centers that are predominantly IT power consumers, or enterprise buildings where human comfort (HVAC) is the dominant consumer of electrical power.

We use the CalIT2 [1] building on the UC San Diego campus as motivation with six floors, open work environments, server room, fabrication facility, and lecture auditoriums. The automation of HVAC and lighting systems in the building is done using a combination of static policies and events inside the building. Building security is automated, but those are mostly set with static policies.

CalIT2's lighting system is currently managed from a single computer. These lights include hallway lights, lab lights, and exterior building lights. However, each light cannot be individually controlled due to an artifact of the underyling electrical wiring. They can only be controlled as a cluster. Furthermore, the lights currently in use cannot be variably controlled. They are either ON or OFF. The current lighting policy policy is to leave them ON from 4:30AM to 10PM, and OFF at other times. Alternate policies can be set based on circuit load values to deal with energy supply problems. During night and weekends, motion and infrared sensors are used to automatically turn on lights when humans are present and light switches can be programmed for override-on for a specific time, usually for 1 hour [6]. However, this is not always ideal particularly for people working at desks for long periods of time.

The HVAC system includes energy inputs from both electrical and mechanical sources. Besides electricity input from the campus power grid, the building is also on a closed chilled/hot water system loop through the campus that uses heat exchangers. A single computer is used to control all aspects of the water system once it enters the building. The chilled water supply system delivers 44°F water, while the hot water supply system delivers 180°F water. As the water is pumped into a particular temperature controlled zone, a variable air volume (VAV) controller nearby controls the flow of the air duct. The water changes the temperature of the air, which in turn heats or cools the zone. If the zone needs to be heated, the VAV controller reduces the air flow to allow the heat from the hot water to diffuse into the air and hence heat the zone. The VAV increases the air flow and shuts off the hot water valve if the room needs to be cooled. Several optimizations can also be made regarding the air flow. For example, an economizer can be used to bring in outside air if it is beneficial. The temperature parameters of each zone are scheduled from a central computer, but can be tuned 2 degrees from individual thermostats. It cannot be tuned more because that would begin to impact other zones [6].

Through just heating and lighting, numerous dependencies and possible inefficiencies exist. For example, temperature zones are mapped to multiple offices, and server rooms and labs have different temperature requirements from the rest of the building. While thermostats and VAV controllers (actuators) can be used to augment the zone, the building is still heated and cooled through the main water pumps. Another dependency is when actuators affect different parameters. For example, an economizer bringing in outside air has the inadvertant effect of also bringing in particulates. A heater continuously maintaining a high temperature has the side-effect of reducing the humidity.

## 3.1 Example Applications

There are numerous actuators for controlling energy use at CalIT2. For controlling the temperature alone there are water and air valves that can be applied separately at different temperature zones. Let us assume for discussion that we can abstract each of these actuators into a single environmental variable. For example, a specific temperature range can be mapped to a specific hot water valve and air flow range. This is not unrealistic since these values are currently manually set to achieve a desired temperature. Thus from an application perspective, the available actuators are: *temperature*, *light*, and *computation*. The available sensors are: *human presence*, *circuit load*, and the actuator variables. Based on these actuators and sensors, we give example applications that could be run in CalIT2.

**Openspace Environment:** This application controls temperature and lighting for employees in openspace work environments. These are specified by the occupants and can vary depending on which occupants are in a given area, or if it is unoccupied.

**Server Room:** This application controls how many machines are active based on the temperature. It makes several simplifying assumptions. It assumes that all machines are physical and not virtual, and that all machines and workloads are homogeneous. If the temperature becomes too high, it will start shutting down machines. The application will periodically try to set a low temperature to maximize computing availability. If it is not able to set a low temperature, it will progressively allow higher temperatures, but shutdown computers.

**Low Energy Handling:** This application ensures that application preferences do not cause excessive circuit load. If current settings cause available energy levels to fall below a threshold, conservation can be done. For example, the temperature requirement can be relaxed to take advantage of ambient air temperature.

## 4. DESIGN

We have built a programming environment, CAhoot, to enable specification and use of actuator requirements and to reduce actuation overheads. This is a preliminary proof-of-concept implementation. It is currently not designed to work in a network setting. Our key idea is that using the range of a monitored variable provides an *application-level intent* that can be shared among multiple applications. CAhoot does not make any policy decisions regarding how the actuators should be used. It merely communicates an application's intent of actuator use and sets an actuator value that satisfies all applications if possible. CAhoot first makes the assumption that application developers are concerned with the results of the actuator rather than programming the actuator itself. For instance, applications typically specify the temperature rather than actuate the heater directly. In the CalIT2 example, a temperature application would know how to actuate the various components to achieve a desired temperature. These values often do not need to be precise. Thus it is possible for applications to concurrently set multiple actuator ranges provided that there is overlap. We define the *actuator range* to be the range of values for a particular sensor actuator that an application wishes to set.

CAhoot introduces an *actuator lock* for dealing with actuator ranges. By adding actuator range information to traditional locks, we can create a partially shared write lock. When actuator ranges overlap, the intersection of the two ranges is used to satisfy both applications. As actuator ranges continue to shorten over time and applications will inevitably have non-overlapping actuator ranges. As a result, CAhoot must also track lock holders, in addition to actuator range information, so that they can be signalled when conflicts occur. Formally, CAhoot allows applications to set actuators with the following properties:

**Atomicity:** Applications need to be able to atomically actuate multiple sensor actuators. Either all actuators are set or none at all. For example, a human comfort application might choose to increase the temperature range only if it can set higher humdity to prevent the air from drying out.

**Range Abstraction:** To allow applications to share actuators more effectively, applications specify value ranges instead of using actuator-specific commands. For example, a human comfort application can tolerate a wider range of temperatures than an application for a fabrication lab.

**Priorities:** When an application requires low latency or when two different applications conflict on the range of an actuator and no compromise can be made, there must be an unambiguous way to

| Application Calls |
|---|
| beginTransaction() |
| setActuator(actuatorId, priority, min, max) |
| endTransaction() |
| getPriority(actuatorId) |
| getMin(actuatorId) |
| getMax(actuatorId) |
| **Application Events** |
| conflict(actuatorId, min, max) |
| changed() |

**Table 1: CAhoot application calls and events.**

decide a range setting. For example, safety applications should have precedence over energy-saving applications.

Table 1 presents the interface that applications use to control a particular actuator. It abstracts the underlying actuator through an actuator range. If a precise value is needed, the range consists of two identical values. In order to set a new actuator range on a sensor actuator, the application must begin a transaction (a term we borrow from databases primarily to indicate atomicity). While a transaction is in progress, other applications cannot attempt to change the range of any actuator. Note that actuator locks are not used for making transactions atomic. Actuator locks are for managing actuator ranges after they are set. Once the transaction begins, the application gives new ranges and priorities for every actuator it wishes to set. A return value is given for each actuator whether the new actuator range could be satisfied. When the application ends the transaction successfully, all actuator ranges (or the respective intersections) become set. This implicitly acquires the proper actuator locks for the application. The priority allows conflicting actuator ranges to be resolved. However, CAhoot leaves the semantics of priority levels to the application domains. Applications setting actuators with equal priority values are processed with a FIFO policy. Actuator locks can implicitly be "released" when applications choose to set an actuator range of $(-\infty, \infty)$. CAhoot also provides an interface for simply reading values.

The last two functions of the CAhoot interface are events that signal applications. When an application attempts to set an actuator range that conflicts with the existing range, a notification is sent to the application with the strictest actuator range[1]. The notified application then has a chance to relax its requirements if it can, but it is not required to. A notification is not sent to every application because only the application with the strictest actuator range can change it. However, when an application successfully sets a new actuator range, every application must be notified because applications may depend on actuator values for correctness. Note that CAhoot does not provide any protection against malicious applications. If applications make very restrictive actuator ranges at the highest priority, then they can lock use of the actuator.

[1]If the application iself has the strictest actuator range, in which case the new actuator range will succeed.

```
START:
if(beginTransaction())
  setActuator(TEMP, 2, 0, 40)
  setActuator(COMP, 2, 90, 100)
  if(!endTransaction())
    // try with relaxed constraints

conflict(ACT_ID, X, Y):
  if(ACT_ID == TEMP && X > 40)
    if(beginTransaction())
      setActuator(TEMP, 2, 0, 50)
      setActuator(COMP, 2, 70, 80)
      if(!endTransaction())
        // try with relaxed constraints

changed(ACT_ID):
  if(ACT_ID == TEMP && getMin(TEMP) > 40)
    // similar to conflict code
```

**Figure 1: Server room application code.**

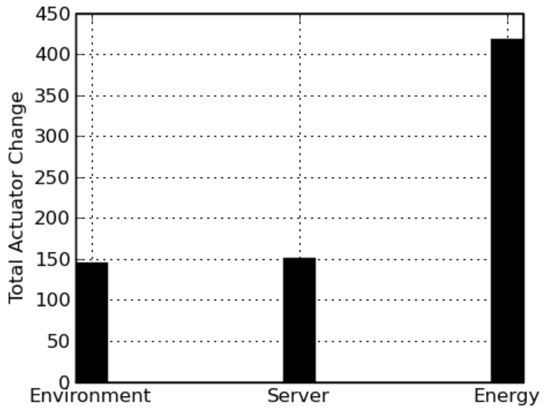

**Figure 2: Total number of actuation actions used as applications are added in order given.**

As an example, figure 1 shows a code snippet of the server room application using CAhoot functions. In this simplified example, we assume only this application will control the the computation amount. If not, event handlers can be written for that as well. The application attempts to set the temperature between 0°C and 40°C and compute capability at 90% to 100%. If it cannot, it will try with a more relaxed set of constraints. If another application attempts to change the temperature, it will try to relax the constraint, but decrease compute capability. Similarly, if it detects the temperature range has changed, it will check to see it requires changing compute capability.

## 4.1 Initial Results

We run the applications in simulation based on preliminary building occupancy data from the ACme building energy measurement deployment [11]. For our evaluation, we run the simulation from 9AM to 6PM and use the previously stated example applications. While this is a work in progress and we do not have the data on actual energy savings, the results show the value of CAhoot in reducing the number of actuator actions. Figure 2 shows that applications can conserve actuation usage in certain scenarios. The Energy application has a much higher sampling frequency, which contributes to its actuator usage count. The Environment and Server applications were able to share a common temperature range.

## 5. FUTURE WORK

We have presented an API that enables a better specification of sensor-actuator actions, and its use in cooperative control. A number of issues and challenges remain to be addressed:

**Deployment:** CAhoot is not currently deployed in a real building. We plan to eventually use it automate building energy use in CalIT2, but questions remain how to deploy and evaluate it in a "live" building, as well as what kinds of savings can be achieved with CAhoot.

**Enforcement:** CAhoot currently does not enforce any fairness. It requires that all applications are cooperative. A malicious application can set narrow ranges at the highest priority.

**Multiple Actuators:** Situations exist where there are multiple localized actuators, such as with lighting control. In this case, specific actuation zones should be specified in addition to the actuator ranges.

**Actuator Dependencies:** There exist acuators that may not be abstracted into a single range. For example, a heater may unexpectedly dry the air as it heats it. An intelligent actuator manager, however, may know this and adjust a combination of actuators rather than a single one.

**Actuator Latency:** Not all actuators can be instantaneously set, which can affect application requirements. For example, temperature is not a variable that can instantaneously change.

## 6. REFERENCES

[1] CalIT2 at UCSD (Brochure). `http://www.calit2.net/newsroom/resources/brochures/pdfs/UCSD_Calit2_brochure2005.pdf`.

[2] GMA Fact Sheet H1N1 Influenza A ("Swine Flu"). `http://www.gmaonline.org/science/swineflufacts.pdf`.

[3] OSHA Ergonomic Solutions: Computer Workstations. `http://www.osha.gov/SLTC/etools/computerworkstations/wkstation_enviro.html`.

[4] TinyOS. http://www.tinyos.net.

[5] *Intel Core 2 Duo Processor E8000 and E7000 Series Datasheet*, 2009.

[6] Tim Beach (CalIT2 Facilities Manager at UCSD. Personal Communication, 2009.

[7] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, 1996.

[8] Department of Energy. Buildings Energy Data Book. `http://buildingsdatabook.eren.doe.gov`, March 2009.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[10] R. K. Harle and A. Hopper. The potential for location-aware power management. In *UbiComp '08: Proceedings of the 10th international conference on Ubiquitous computing*, pages 302–311, New York, NY, USA, 2008. ACM.

[11] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity ac metering network. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN 2009)*, 2009.

[12] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM.

[13] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[14] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 211–224, New York, NY, USA, 2008. ACM.

[15] A. S. Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001.

[16] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 123–132, 2002.